Syntax-Directed Editing: Towards Integrated Programming Environments

Raul Medina-Mora

Department of Computer Science
Carnegie-Mellon University
Pittsburgh, Pa. 15213

March 1982

DEPARTMENT
of
COMPUTER SCIENCE

Carnegie-Mellon University

Syntax-Directed Editing: Towards Integrated Programming Environments

Raul Medina-Mora

Department of Computer Science
Carnegie-Mellon University
Pittsburgh, Pa. 15213

March 1982

Submitted to Carnegie-Mellon University
in partial fulfillment of the requirements
for the degree of Doctor of Philosophy

This work was sponsored in part by the Software Engineering Division of CENTACS/CORADCOM, in part by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 3597, monitored by the Air Force Avionics Laboratory Under Contract F33615-81-K-1539, and in part by the National University of Mexico.

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the US Government.

Para Katty y Katina, con todo mi amor

Acknowledgements

It is a great pleasure to acknowledge the untiring support of my advisor, Nico Habermann. Throughout these years, he always gave me the necessary confidence to keep going. His contribution to the organization and completeness of this thesis is invaluable. I would also like to thank the other members of my thesis committee - John Nestor, Eugene Ball and Tim Teitelbaum - for their helpful comments and criticism that in many ways improved the quality of this document. Special thanks go to David Notkin for carefully reading the various drafts of the thesis and suggesting many improvements. He and Peter Feiler spent many hours discussing and improving a good number of the ideas presented in this thesis. They and other members of the GANDALF project at CMU - Robert Ellison, Gail Kaiser, David Garlan, Phil Wadler and Steve Popovich - provided invaluable help in the design and implementation of various parts of the ALOE system. I would also like to thank James Gosling who wrote a very good display package for his UNIX[tm] EMACS system that was used for the ALOE system.

There are no words to describe the encouragement and support offered by Katty, my wife, that helped me enormously in getting through the arduous task of writing this dissertation. She and Katina, my daughter, made the whole experience of these years a very worthwhile one.

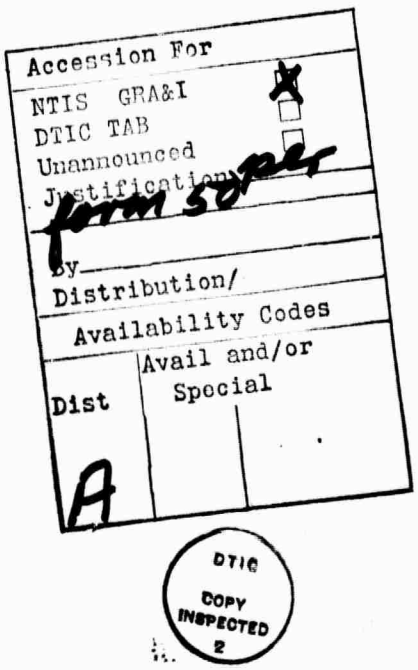

Table of Contents

List of Figures

Chapter 1

Introduction

1.1. Software Development

One of the major goals of software engineering is to enhance programming quality and productivity, in particular, for producing large software systems. Programming language design and methodology have contributed towards this goal by developing and supporting concepts such as modularization and data abstraction and encapsulation. The MESA [Mitchell 78] and ADA [DoD 80] programming languages are examples of the evolution of these concepts. With these concepts, programming languages can support the static aspects of program construction by letting the programmer, among other things, describe the modular interfaces of his system, localize the decisions of representation, *etc*. Programming languages lack support for the dynamic aspects of the development and maintenance of large software systems: system developers must deal with different versions and compositions of the different pieces of a system, as well as with the interaction of several programmers in the development of a system.

Programming environments may provide the necessary means to address the problems associated with the dynamic aspects of development and maintenance of large software systems. Programming environments have traditionally been considered to address only the problems of the development of single programs by single users [Donzeau-Gouge 80, Teitelbaum 81a, Ritchie 74]. Environments that in addition, address the dynamic aspects of the development and maintenance of large software systems are often referred to as software development environments [Habermann 82, Deutsch 80]. These environments provide facilities for system version control and project management which must be expressed through some language [Kaiser 82]. Environments can be built in which the facilities for dynamic system development and maintenance can be provided through the

Interaction between user and system. Since users will always be editing some language to accomplish their task, it being writing a program or building a system, we will use the term programming environments to refer to all these environments.

Integrated programming environments provide a single environment through which the programmer can accomplish his task instead of having to construct and maintain his system using a set of independent and unrelated tools. The dynamic aspects of program development will be addressed through the active participation of the programming environment in the development of such systems. The tools of an integrated programming environment understand each other's functions and objectives and can collaborate towards a common goal. The integrated environment has knowledge about the objects that it manipulates and their current state. It is therefore able to respond to incorrect or undesirable user actions. A common internal representation provides the means of communication among the tools.

Language-oriented programming environments are knowledgeable of the language and thus can help and cooperate with the programmer in his task. In the process of creating, modifying and executing programs and systems, the programmer is able to concentrate on his algorithms and on the structure of his systems, instead of on their specific textual representation.

In recent years, there have been some important developments in computer technology that make possible the development of these new systems. Better displays and terminals make possible the development of sophisticated display-oriented systems. With the development of more powerful computers, more computing cycles are available for interactive systems to perform computations between interactions with users and during users' *think*-time at the terminal.

One possible way of exploiting the features of these new technology developments is by enhancing the current methodology for software development. This methodology typically consist of a set of unrelated tools: text editor, compiler, loader, debugger, *etc*. Each one of these tools performs part of the programming task. This approach has lead to the development of very sophisticated display-oriented text editors such as EMACS [Stallman 81] and UNIX™ EMACS [Gosling 81a]. Within the programming task, these text editors are still editing text files and their interaction with users is in terms of characters, words, lines, *etc*. A

very limited amount of knowledge can be incorporated to provide certain expansions of program constructs and to provide formatting of programs automatically. However, there is no knowledge of the syntax and semantics of the language to insure the correctness and consistency of programs nor to provide immediate feedback on errors.

UNIX EMACS goes further than just text editing by providing access to other tools through its interface. For example, the user can invoke a compiler to process his program, UNIX EMACS processes the error messages produced by the compiler and positions the user in the line of the file where errors occurred. However useful this features may be, they do not give UNIX EMACS the functionality of an integrated environment in that the tools of the environment do not have a common knowledge of each other and thus do not collaborate towards a common goal.

A different approach is to combine a different methodology with the new technology. This different methodology is that of language-oriented editors, an idea demonstrated for the first time in the Emily System [Hansen 71]. Language-oriented editors interact with the users in terms of the language constructs and, in addition, can guarantee the syntactic correctness of programs and systems. Language here is meant in a wide sense, it is not limited to programming languages but to any language or structure that can be expressed with a context free grammar. If semantic knowledge can be incorporated, these editors can ensure semantic correctness and consistency or at least provide immediate feedback on errors. This implies that these editors actively participate with the programmer in the development of his programs and systems. The language in a large integrated environment can include, in addition to the programming language, descriptions of system structure such as the structure of its components, versions of subsystems, documentation, *etc*. Other tools of the environment also have knowledge of this language and it is therefore possible to integrate them with the editor to form an integrated environment.

Syntax-directed editors are language-oriented editors in which the programs and systems are created and modified according to the syntactic structure of the language instead of characters and lines. Some of the important characteristics of syntax-directed editors that will be considered in this thesis are the following:

- A constructive approach to program and system building. The programmer concentrates on constructing his programs and systems as structures rather than as collections of pieces of text. Programs and systems are built through constructive commands. Each command corresponds to a construct of the

language. The burden of generating the concrete syntax (*i.e.* syntactic sugar) is taken over by the editor and is automatically generated.

- Manipulation of programs and systems in terms of their structure. The focus of attention for all activities of the programmer is always a construct of the language. The programmer operates on his program or system in terms of their structure and not through their textual representation.

- Abstract syntax trees. Programs are represented internally as syntax trees that are built by the editor from constructive commands issued by the programmer. These trees are abstract syntax trees, whose nodes represent the language constructs. The elements of the concrete representation of languages, such as keywords, punctuation marks, separators, terminators, *etc*, are not kept in the tree.

- Syntax tables. If an editor is written for a particular language, the syntax information is part of the implementation of the editor. In syntax-directed editors generated from language descriptions the syntactic knowledge of the languages obtained from these descriptions is kept by every editor in a collection of syntax tables. The editors use these tables to ensure the syntactic correctness of the programs being built.

- Unparsing. The concrete representation of structures displayed in the screen is obtained by unparsing the internal abstract syntax tree representation into a visual form. Associated with every construct of the languages, a set of rules, called *unparsing schemes*, specify the visual representation of the construct.

- Uniform interface for the environment. The user interface of a syntax-directed editor provides the means for communication between user and environment. The user communicates with all the tools of the environment through the user interface of the editor. As the environment is knowledgeable of the development task, some of the tools are automatically applied at the appropriate times.

1.2. Goals of Thesis

The purpose of this dissertation is to investigate the design and implementation of syntax-directed editors, the ability to generate and extend them, the appearance they give to the user, the properties of languages for which such editors are generated, the capabilities and the limitations of the available hardware and display devices. More specifically, this dissertation addresses the technical issues involved in:

- The design of syntax-directed editors to provide a uniform user interface for integrated environments.

- The automatic generation of syntax-directed editors from language descriptions, which was first attempted in Emily [Hansen 71] but has not been an important goal in other systems.

- The extension of syntax-directed editors to understand language semantics (*i.e.* perform context sensitive processing), through a flexible procedural approach instead of through some formal specification [Demers 81].

- The extension of syntax-directed editors to support the development of large integrated environments that support the dynamic aspects of the development and maintenance of large software systems. In other systems [Donzeau-Gouge 80, Teitelbaum 81a] the approach focuses on the environment for a single programmer working on a single program.

In order to address and understand these technical issues, a generator of extensible syntax-directed editors has been designed and implemented. Throughout this dissertation we will refer to any such editor as an ALOE (A Language Oriented Editor).

1.3. Major Design Issues

In the design of syntax-directed editors, some of the most important technical issues that arise are:

- User interface. The user interface of syntax-directed editors is extremely important. The usability of the editor depends, in a significant manner, on its user interface. The user interface of a syntax-directed editor is very different from that of a text editor. Most users like the flexibility provided by good text editors to edit their programs. For the user to accept this new methodology, the user interface of a syntax-directed editor must be *friendly* and simple.

- Display-oriented functionality, hardware resources and limitations. Proper exploitation of new display technologies can have a great impact on the functionality of an interactive system. On the other hand, there is a wide variety of equipment available, and design decisions must be made based on the actual hardware resources and limitations.

- Separation of abstract syntax and concrete representations. One of the most important characteristics of the design of the grammatical description used to specify the languages, is the separation of the abstract syntax of the language from its concrete representation. This separation makes it possible to produce multiple concrete representations for every construct of the language. A large part of what is understood as the syntax of a language is not part of the abstract syntax but of its concrete representation (punctuation marks, keywords, separators, *etc.*).

- Common internal representation. The abstract syntax trees built by the syntax-directed editor are the common language representation used by all the tools of the environment. This common internal representation provides the means of communication among the tools of an integrated environment.

- Action Routines for context sensitive processing. In order to support an integrated programming environment it is necessary to provide the means for context sensitive processing such as checking programming language semantics.

- Language or Environment specific behavior. For a successful programming environment, certain functions must be *tuned* to the specific environment. Symbol table manipulation is an example of one such function. Different languages and environments need different functionality for their symbol table manipulation. It should be possible to provide this needed functionality.

- Generic Systems. It is different to build a syntax-directed editor for a particular language than to build a generator of such editors. Some design decisions are influenced by this difference. In particular, design decisions must be made to provide solutions with general mechanisms rather than providing a specific solution that solves a problem for one language but not for another.

1.4. Previous and Related Work

The ideas of syntax-directed editing and integrated programming environments are not new, they have already appeared in several systems. Some of the most important efforts include:

- The Interlisp System [Teitelman 78] is a very sophisticated programming system for LISP. The simple syntax and semantics of LISP lend themselves very well to more structured manipulation of programs, its interpretive nature lends itself better to the edit/interpret approach. Interlisp incorporates powerful facilities like structured editing, sophisticated debugging techniques, automatic error correction, the programmer's assistant and others. In this thesis we will address the issues involved in the development of environments for a variety of languages with more complex syntax and semantics.

- The Emily System [Hansen 71] was one of the earliest efforts with syntax-directed editing. It is a menu-driven system in which the user constructs a program by selecting BNF productions [Backus 59]. The BNF productions include the concrete representation of the language constructs. Emily is not an integrated system: in order to compile a program, text is produced and has to be compiled separately. Performance in Emily was not very good, there was a noticeable delay in the update of the screen after every interaction. For the generation of ALOES, a grammatical description that emphasizes the language constructs themselves and not some productions of certain grammar, is proposed. The grammatical description also makes a clear separation between abstract syntax and concrete representation. We want to address the support of integrated systems through the sharing of a common program representation and a uniform user interface. Current developments in computer technology make it possible to enhance the performance of interactive systems.

- MENTOR [Donzeau-Gouge 80] is a structured editor for PASCAL. The user writes his program as text and MENTOR parses it to create the syntax trees. The user then manipulates his programs structurally. Editing commands and context sensitive routines are written in MENTOL, a tree manipulation language that is the command language of MENTOR. The context sensitive routines are invoked explicitly by the user. MENTOR is not an integrated system either: a program must be unparsed into a text file before being compiled separately. MENTOR is not a display-oriented system, it uses a one-dimensional scroller interface. ALOE follows a structural approach for entering, as well as for editing structures. Context sensitive routines are written is C [Kernighan 78] using an implementation environment that includes all the primitives for tree manipulation. The context sensitive routines are automatically invoked at the appropriate times. ALOE supports integrated environments and has a display-oriented interface.

- The Cornell Program Synthesizer [Teitelbaum 81a] is written for a particular language, PL/CS, a small subset of PL/I. The Synthesizer builds code trees that can be interpreted and unparsed. The Synthesizer achieves a high degree of interaction between program construction and execution, in an environment for a single programmer building a single program. The Synthesizer is a hybrid system that manipulates programs structurally at the statement level and textually at the expression level. In this thesis, we address the issues of generating syntax-directed editors for a variety of languages, with structured editing at all levels of the language. We also address the development of environments that support the dynamic aspects of system development and maintenance for large software systems with interactions of several programmers.

Some of the important distinguishing characteristics of an ALOE, as discussed in this dissertation, include:

- An ALOE is an editor generated from a grammatical description instead of being hand crafted for a particular language.

- The grammatical description separates the abstract syntax structure and the concrete representation of the language constructs, which permits the specification of multiple concrete representations for every language construct.

- Structured or constructive editing is performed at all levels of the language, there is no parsing performed.

- ALOEs have the ability to perform context sensitive processing implicitly through action routines associated with every construct of the language.

- Environment specific functionality can be added through the redefinition of certain environment dependent functions.

- The functionality of ALOEs is intended for the development of large integrated environments, the internal representation is available as the common representation for other tools of the environment, and it provides the means of

communication between these tools. The user communicates with other tools of the environment through the uniform user interface of ALOE. The integrated environment approach has a significant impact on the overall performance of the system because duplication of effort is avoided and the information about the programs is preserved and does not have to be recomputed.

- We tend to emphasize the generation of ALOEs for programming languages and systems because they are their motivating application, but ALOEs can be generated for all structures and languages that can be expressed using the grammatical description. An important example is the ALOE generator itself which is an ALOE in which the user edits language descriptions and that produces the language tables through the use of several unparsing schemes. Other possibilities include a mail system [Notkin 82a] and an ALOE for SCRIBE [Reid 80] a document production system. For this reason, syntax-directed editing can also be referred to as structured editing.

1.5. Structure of Thesis

The organization of this dissertation is based on the motivation and discussion of the major design issues, and on the evaluation of the corresponding design decisions. So, for example, the issue of separation of abstract syntax and concrete representation is discussed in terms of its impact on the user's view of the system; how it affects the generation of ALOEs; its impact on the implementation of action routines; and its role in the development of large integrated environments. In the process of discussing this issue from these different points of view, some of the arguments and motivation may be repeated. The other major design issues, listed in section 1.3, are similarly discussed.

Consequently, in chapter 2, the characteristics of the user interface and the design decisions involved are discussed. In chapter 3, we address the issues involved in the generation of syntax-directed editors. In chapter 4, the facilities to provide context sensitive processing and manipulation of the environment are discussed. In chapter 5, we discuss the issues that arise in the development of large integrated environments. In chapter 6, the design decisions are evaluated and a comparison is made with UNIX EMACS [Gosling 81a] and with the syntax-directed editors and environments mentioned in the previous section.

Throughout the thesis, the emphasis is placed on the motivation of the design decisions, the discussion of alternatives and their evaluation. Detailed description and explanation of these aspects is not included in the main body of the thesis. Some of the necessary descriptions are included in the following appendices: appendix A gives a list of the editing

commands common to all ALOEs. Appendix B describes the commands available for use in the unparsing schemes. Appendix C lists the specification of the routines of the ALOE implementation environment, which is discussed in chapter 4.

A complete detailed description of the ALOE system is provided in the *ALOE Users' and Implementors' Guide* [Medina-Mora 81a]. The guide is structured as follows: chapter 1 gives a short introduction. Chapter 2 is the ALOE Users' Manual. Chapter 3 describes the ALOE Generator and the grammatical description rules. Chapter 4 describes the implementation of the internal representation. Chapter 5 explains the interface to action routines. Chapter 6 defines the interface to extended commands. Chapter 7 provides an operational definition of the internal structures through a set of routines available to the implementor of an ALOE which can be invoked from action routines or extended commands. Chapter 8 defines the system or environment specific routines whose standard implementation is provided. Any implementation can redefine some or all of these routines. Chapter 9 describes the window manipulation package and how to use it. Chapter 10 describes the use of the display handling package. Chapter 11 explains the process of generating an ALOE once the grammar has been created and the rest of the system has been written. Finally, appendix I gives a detailed example of the implementation of an ALOE for a simple language. The example defines action routines, provides extended commands, and redefines some of the environment specific routines.

Chapter 2

User Interface

2.1. Goals of the User Interface

One of the most important goals of the user interface of a syntax-directed editor is to make it possible for the user to interact with the editor in terms of the structure of his programs and systems. Other important goals include:

- Minimization of effort. It should take the user minimal effort to indicate the actions he wants applied. Especially, it should be very easy to invoke simple and frequently applied operations.

- Flexibility. The user interface should be flexible enough to respond to the needs of novice as well as expert users.

- Immediate feedback to user. In an interactive environment, the user should know the current state of his system. This can be achieved by updating the screen after every interaction. In this manner, the user gets immediate feedback as to the effect of the action he just invoked.

- Minimization of delay. Any interactive system should provide an adequate response time. Delays between interactions should be small.

2.2. Syntax-Directed Editing

One of the important advantages of a syntax-directed editor as the uniform interface of an integrated programming environment is that, in the process of creating, modifying and executing programs and systems, the programmer is able to concentrate on his algorithms and program and system structure, rather than on their specific textual representation and on the different tools of the environment and the different languages used to communicate with them.

An ALOE can be visualized as a constructive editor that understands the syntax of the language. Language constructs, such as variables, operators, expressions, statements, declarations, *etc.*, can be added, modified and removed. The user communicates with ALOE in terms of these language constructs. The user constructs his program by inserting templates representing different language constructs and then filling the *unexpanded* parts of those templates with other templates. Since ALOE knows which constructs are valid at any given point, it allows the programmer to insert a language construct only where it is syntactically correct.

In its interaction with an ALOE the user thinks of his program or system in terms of the language constructs and lets ALOE handle the concrete syntactic details. For example, when a user adds an `if` statement to his program, he thinks of the abstract structure of the construct: a language construct with a test and two statements to be executed depending on the outcome of the test. ALOE adds all the keywords, punctuation marks, separators, terminators, *etc.*, required by the language. So, instead of typing the character sequence for an `if` statement in C [Kernighan 78], the user calls on the template `if`.

Figure 2-1 shows the display before the construction of the `if` statement and the resulting display after its creation. The program cursor, highlighted in the display (indicated by a rectangle in the figure) is advanced to `<expression>` so that it can be similarly expanded. This construction was permitted because it was syntactically correct, the program cursor was located at a `<statement>` meta node indicating that only a `statement` could be legally constructed at that point. Note that ALOE provides all *syntactic sugar* required by the language like the parentheses around the `<expression>` construct required by the C language syntax. Problems such as misspelled or non-matching keywords cannot occur because the language constructs are inserted by ALOE and not by the typist. ALOE relieves the user from worrying about the concrete syntactic constraints imposed by the language.

Internally ALOE represents the program as a *syntax tree*. Each template corresponds to a node of a certain type in the tree. The unexpanded nodes of the template are the offspring of the node. These unexpanded nodes will be referred to as *meta* nodes. They will be filled in with subtrees representing their expansion. Terminal nodes correspond to the terminal operators of the language (variables, constants, *etc.*). Non-terminal nodes correspond to non-terminal operators and are of two types: fixed arity nodes with a fixed number of offspring and lists with a variable number of offspring. For a complete description of the internal representation of ALOE see [Medina-Mora 81a].

```
struct fract *add(struct fract *fr1, *fr2)
{
    struct fract *fr3;
    <statement>
    return(fr3);
}
```

```
struct fract *add(struct fract *fr1, *fr2)
{
    struct fract *fr3;
    if ( <expression> )
        <statement>
    else
        <statement>
    return(fr3);
}
```

Figure 2-1: Construction of an IF statement

The user actually constructs and manipulates a program tree without necessarily being aware of it. However, it is desirable for the user to think of the program in terms of the syntax tree instead of the displayed text. To display the text of a program to the user, ALOE uses an *unparser* that translates the syntax tree into text. As part of its task the unparser formats the program. The unparsing process is driven by *unparsing schemes* specified in the grammatical description. These schemes describe the mapping from the abstract syntax to the concrete representation for every language construct. The clear separation of abstract syntax and concrete representation places the emphasis on language constructs and not on specific syntactic details.

Many design decisions of a user interface are determined in a significant manner by the characteristics of the input and output devices chosen for an implementation, as well as by the baud rate of the communications line. The design space of input devices includes keyboards with different characteristics and different kinds of pointing devices. Keyboards can include special characters (*e.g.* control characters, meta characters, *etc*), function keys, cursor pads, *etc*.

The design space of output devices ranges from simple scroller interface (hardcopy devices and display devices used as such), to the use of displays as two-dimensional devices, to sophisticated graphic displays. Display's characteristics can also include several fonts and highlighting capabilities.

For the implementation described in this dissertation we have chosen the Concept-100 terminal [HDS 79]. Its keyboard includes control characters, function keys and cursor pad. The display is used as a two-dimensional device. It includes highlighting capabilities as well as mechanisms, such as character and line insert/delete, which facilitate the fast update of the screen.

2.3. Area Cursor

To provide immediate feedback to the user, at any point in the interaction ALOE displays the current state of development of the program. The program cursor, *i.e.* the position in the program tree where the next command will be applied, must be indicated in the display. Taking advantage of terminal hardware facilities, ALOE uses an *area cursor*. ALOE highlights (in reverse video, for example) the textual expansion of the whole subtree. This mechanism also helps in emphasizing the structure of the program and provides an excellent feedback to the user: when at a statement, the whole statement is highlighted, when at an expression, the whole expression is highlighted.

Some systems, such as the Synthesizer [Teitelbaum 81a] use a single character cursor to indicate the program cursor. In some cases a single character is not enough to indicate the current program structure that the cursor represents: it is ambiguous. In the Synthesizer there is only one case of ambiguity of the single character cursor. It is described in section 6.8.1, but this is not necessarily the case in general. In ALOE, the cursor can be placed at a list to refer to the entire list. In this case the single character cursor would be ambiguous because the cursor would be placed under the first character of the first element of the list and it would not be clear if it refers to that element or to the entire list. This situation does not arise in the Synthesizer because the cursor can only be placed at elements of lists and not at the entire list. When an editing command is applied to a list, the extent of the list must be explicitly specified. When the cursor is used at run time, to trace the program as it is executed, the cursor is changing positions very rapidly and a single character cursor may be better in this case [Teitelbaum 81b]. If the *grain* for tracing is not too small, for example, tracing only statements and not expressions, the area cursor is also acceptable [Feiler 82a].

An alternative would be to use a two character cursor [Barstow 81]: the first and the last character of the subtree expansion. This solves the ambiguity problem but does not necessarily emphasize enough the structural view of the program.

The textual expansion of the internal tree representation is controlled by the unparsing schemes. If a particular operator's unparsing scheme does not include any text (aside from the text generated by its offspring), it could cause some ambiguity in the display. It is up to the ALOE implementor (*i.e.* the writer of the grammatical description for an instantiation of an ALOE), to make sure that all operators have some text in their unparsing scheme. This ambiguity problem is independent of the type of display chosen for the program cursor. It is frequently found in the cases of lists with one element. In general, the list operator itself does not add any extra text to the text produced by the single element. This problem is recognized, and cursor movement, discussed in section 2.4.2.1, takes it into account.

2.4. Command Language

Every ALOE has two different types of commands: *editing* commands and *language* or *constructive* commands. Editing commands are common to all ALOEs. They are used to invoke language independent operations such as program cursor movement, read and write from files, insert, delete and modify subtrees, *etc*. Constructive or language commands are the names of the language constructs (henceforth referred to as *operators* of the language) defined in the grammatical description of every language.

Editing and constructive commands are specified using different naming conventions, so that the ALOE implementor can select good *mnemonic* names for the language operators without having to worry about naming conflicts with the names of editing commands. This also allows for extensions to the set of editing commands independently of any language. Extended commands can be added to an ALOE to implement language or environment specific operations. The interface of extended commands is the same as for editing commands.

For any ALOE command the user only has to type the minimum number of characters needed to specify the command. This design decision helps achieve the goal of minimization of user effort.

2.4.1. Constructing or Language Commands

The names of constructing or language commands are the names of the operators of the language defined in the grammatical description. Figures 2-2 and 2-3 illustrate a sample editing session for the process of constructing a `for` statement in C [Kernighan 78]. The first column gives the commands typed by the user, the second column shows how the terminal display would appear after the command is executed and in the third column the syntax tree structure being built is presented.

When the ALOE implementor is choosing names for the operators of the language, he must be aware that he is designing part of the user interface of the editor, he is choosing the names for the constructive commands too. In the grammatical description, operators are grouped together in *classes*. Every class contains the set of valid operators that can replace a meta node of a particular offspring of a non-terminal node. Within any class the set of operators should not have names with common leading characters, so that the user can type few characters (one would be best) to specify the command, but the ALOE implementor must choose the names so that they are mnemonic (*i.e.* their name indicates unambiguously what they stand for).

Another characteristic of the grammatical description that impacts the user interface is its *flat* structure as opposed to the hierarchical structure of traditional BNF formalisms [Backus 59]. For example, at the expression level, any operator can be applied directly without having to apply intermediate operators such as `term` and `factor` which would be necessary with a hierarchical definition. Precedence values are associated with the operators to be able to generate the correct parenthesization of expressions. One of the early efforts with syntax-directed editing [Hansen 71] uses a modified BNF formalism in which non-terminal operators at the expression level are at the same level, but terminal operators such as variables and constants are not. It is necessary to go through one extra hierarchical level to construct simple expressions such as 'a + b'. A more detailed comparison with traditional BNF formalisms can be found in section 3.2.9.

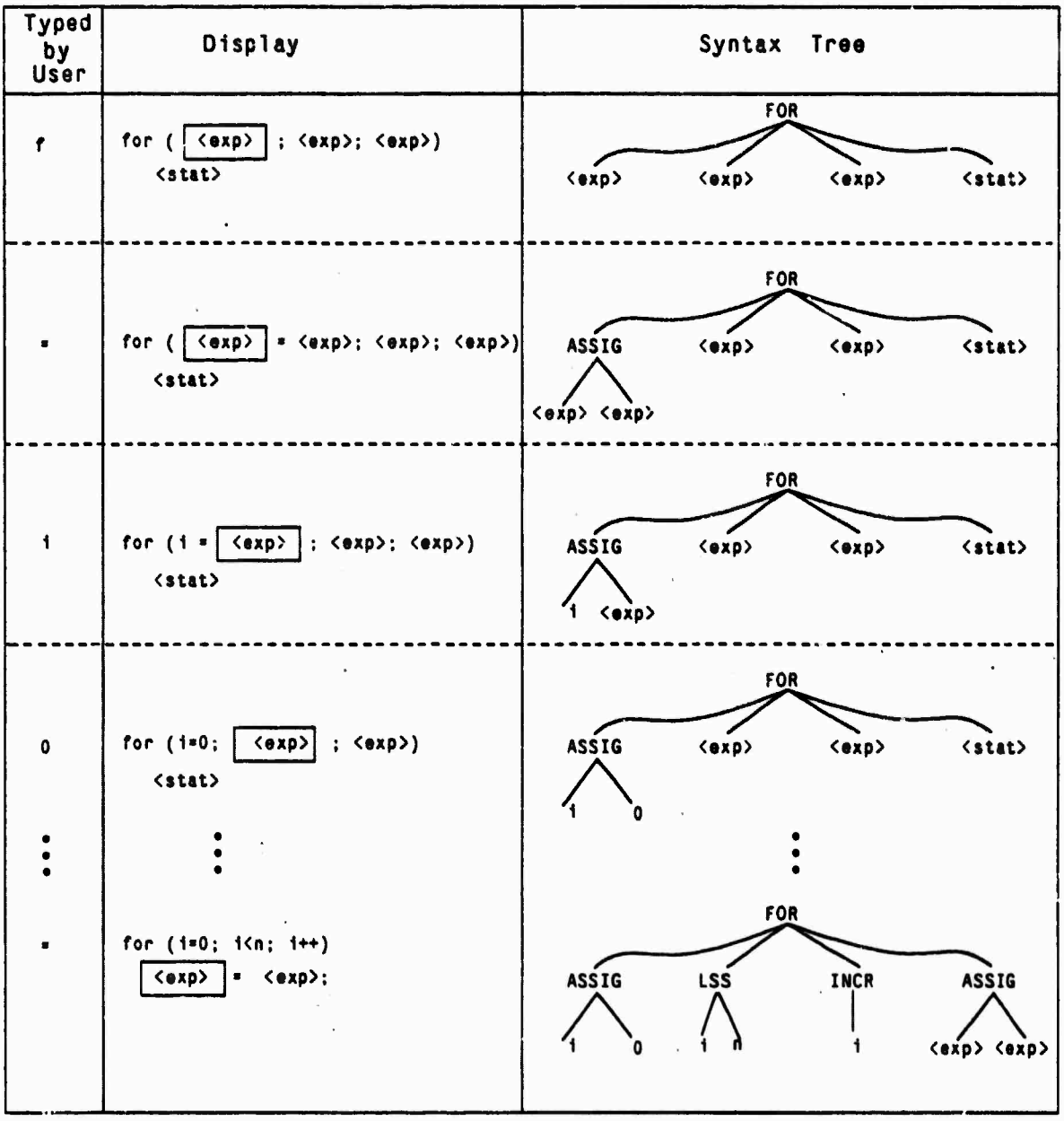

Figure 2-2: A Sample Editor Session

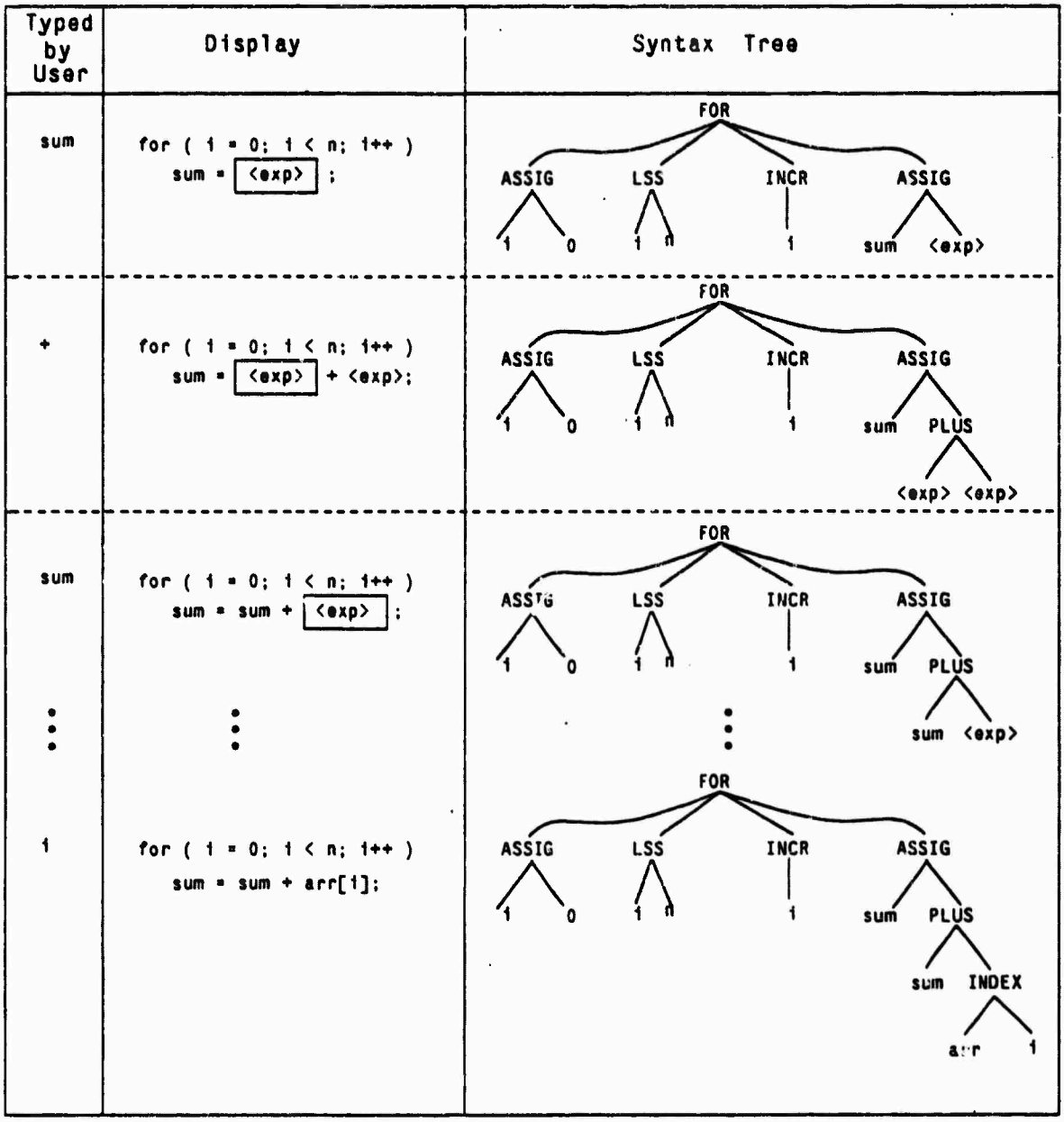

Figure 2-3: A Sample Editor Session (continuation)

2.4.1.1. Synonyms

One important impact on the dual role of grammar operators as constructive commands is that, for some constructs of the language it is *natural* to think of them in terms of their name; such is the case of constructs like `while` or `for` statements. But there are others such as `plus` and `less-than` for which a more graphical representation would be better for a command name. To address this problem the concept of synonyms for language operators was introduced. These synonyms are specified as part of the grammatical description. The user can invoke a command by typing its name or its synonym (the characters '+' and '<' would be the synonyms of the examples cited above). It was found that this addition greatly improved ALOE's user interface. Users thought that it was extremely cumbersome to build expressions using names such as `plus`. After synonyms were introduced users felt that constructing expressions was much easier.

2.4.1.2. Lexical Units

Some terminal operators have values associated with them. For example, the name of a variable, the value of a constant, *etc*. ALOE prompts for these values whenever the user applies the corresponding command.

ALOEs do a very limited form of automatic lexical analysis for terminal operators by distinguishing between variables and different types of constants. Action routines can perform some form of lexical analysis by validating or rejecting the associated values of terminal operators given by the user.

An important improvement can be made to the user interface by making ALOE understand about lexical units. The current set of terminal operator types in the grammatical description includes variables and constants. A better set of constant operator types, such as integer, real and character constants, can be incorporated. A lexical routine would be associated with each one of these types for recognition. The ALOE implementor could redefine any one of these routines to accommodate special language characteristics. For the classes that contain an operator of one of these types (*e.g.* the `expression` class), the ALOE generator could ensure that all operators of the class have a non alphanumeric synonym. When the program cursor is at a meta node that corresponds to such a class, the lexical routines would be invoked to identify a command as one of these constant operators types, if one is successful, the corresponding terminal operator would be invoked and the command string is given as the associated value.

This suggested improvement is equivalent to requiring that the operators of these classes have non alphanumeric names (*e.g.* to have '+' as the operator name instead of PLUS), or that full names be used to specify them (for operators such as MOD and DIV in PASCAL [Jensen 74]). Section 3.2.2.1 contains further discussion of lexical analysis issues. Section 3.2.7 further discusses synonyms for language operators.

2.4.1.3. Screen Update

ALOE updates the display after every command is applied. With bandwidth limitations it is often desired to issue several commands before an update. In ALOE several commands can be typed in a single line of input. ALOE then applies each command, one at a time. The display is updated only after the application of the last command. This also eliminates intermediate states that could be distracting. Drawbacks of this approach are especially evident when there are errors caused by a command and the rest of the commands must be flushed or applied out of context. A better interface would let the user edit the command line after the error, avoiding both problems.

An alternate solution is to provide an explicit command to update the screen. There are two important drawbacks to this approach. First, it requires more effort from the user by having to explicitly invoke this command every time an update is desired. Second, the display will not reflect the current state of the program which is an important goal of interactive systems.

2.4.1.4. Cursor Movement

After every constructive command, ALOE moves the cursor to the next meta node, thus *guiding* the user in the construction process by placing the cursor in the next available node for expansion. During program construction this may be the desired behavior, but during program editing it may not be: if the program is already complete then the result is acceptable, the cursor is kept at the current node, but if there is an unexpanded node in an unrelated section of the program, ALOE moves the cursor to an undesired location, causing confusion to the user. Instead, it is desirable that the cursor be kept in the immediate context.

There are several ways to solve this problem. First there is the .back command that moves the cursor from its present location to the previous one (see section 2.4.2.1) . The second solution is that ALOE interprets the command terminator to indicate the action to take. There

are two different standard command terminators, the normal one (a *CR*) indicates that ALOE should move the cursor to the next meta node, the second one (a *LF*) indicates that ALOE should leave the cursor at the resulting node after the command is applied. There are two important drawbacks to this solution. First, the user will not always remember to type the second command terminator, and second, with multiple commands in one line there is a need for two command separators, which makes the command interface more confusing.

Any cursor movement command can also be used as a command terminator, as is also the case in the Synthesizer [Teitelbaum 81a]. The cursor will then be placed at the corresponding node after applying the cursor movement command. This feature is hardly ever used in practice in ALOE; in the Synthesizer it is important because it provides the transition between text editing of phrases and structured editing: when a user is at a phrase it should not matter if he just entered it or he moved the cursor there, in any case a cursor movement command should be applicable.

The third solution, and possibly the best, is the definition of *constructing* and *editing* modes of operation. While in constructing mode, ALOE moves the program cursor to the next meta node. In editing mode it moves the cursor to meta nodes only within the newly created subtree in case of a construction of a non-terminal operator, and moves to the next node (meta node or not) after the creation of a terminal operator. That is, the search for meta node is restricted to the current subtree. These three solutions are not mutually exclusive. They all are available in any ALOE. Section 2.7 contains further discussion on the different modes of operation of an ALOE.

There can be classes in the grammatical description that contain only one non-terminal operator. When a meta node for these classes is created, ALOE will automatically apply the operator, thus saving the user the need of invoking it explicitly. If the only operator of a class is a terminal operator, automatic application is not performed because this requires the user to specify also the value of the operator (*e.g.* the name of a variable, the value of a constant, *etc.*), and the user may not want to instantiate those terminals yet. Automatic application could be extended to terminal operators that do not have values associated with them (*e.g.* *static* terminals used for predefined type names of a language, such as `int`, `float`, *etc.*).

2.4.2. Editing Commands

Editing commands are used to invoke language independent operations such as program cursor movement, read and write from files, insert, delete and modify subtrees, display manipulation, *etc*. In this section we discuss the interface of editing commands as well as some of the more interesting commands. For a complete list of all editing commands and their functionality see appendix A.

As discussed before, editing and constructive commands are specified using different naming conventions. Editing commands are prefixed with a period ("."). This solution achieves the purpose for which it was designed but does not necessarily provide a good interface for an expert user. The user has to type at least three characters (the dot, at least one character for the command and a command terminator) to specify any editing command. To improve this, the use of ASCII control characters as synonyms for editing commands was introduced, taking advantage of the fact that language commands will not use control characters in their names. Experience with the use of control characters for commands has been good with text editors such as EMACS [Stallman 81]. To experiment with these decisions and to provide the flexibility necessary for different kinds of users, both forms are available in an ALOE: sometimes it is easier for a user to remember a command by its name than by its control character synonym.

Users go through a learning process of the command language. Again, experience with the use of EMACS [Stallman 81] supports this hypothesis: once the user learns the command equivalences, he will not need to refer to their name. To provide the necessary flexibility, a *help* facility displays both names and synonyms of commands. The considerable savings of keystrokes (a command terminator is not needed, neither is the '.') makes the learning effort worthwhile. In EMACS, the user may decide the binding of control character keys to editing commands. Some users may want to decide a binding that helps them remember the commands much easier, or to bind certain commands of their preference (there are more commands than keys available). On the other hand, this causes the creation of many individualized and incompatible editors.

We have used the cursor pad function keys to invoke the program cursor movement commands with great success. We have not yet fully explored the use of other terminal function keys to invoke editing commands. This is a possibility explored with success in some

other systems such as the Synthesizer [Teitelbaum 81a]. The only big problem of using function keys is that any terminal keyboard has a limited set of these keys and this could preclude making extensions to the set of editing commands. Given this limitation, terminal function keys should be used for the most common editing commands.

2.4.2.1. Cursor Movement

Cursor movement is not done following the textual representation of the program (as most programmers are used to with text editors), but rather following the structure of the program, reflecting the result of the move by highlighting the new program cursor. There are five basic cursor movement commands defined in an ALOE:

- Cursor **out**: Moves the cursor one level up in the program tree (to the *parent* node). If the cursor is at an element of a list with only one element, ALOE moves the cursor past the *list node* that represents the whole list. This feature was added to the original design of ALOE because in most cases the highlighted area in the display does not change when moving to the list node, and it was very confusing. The change was found to be extremely helpful: it is always clear where the program cursor is.

- Cursor **in**: Moves the program cursor one level down in the tree. Moves it to the first offspring in a fixed arity node or to the first element of a list. If the cursor is moved into a list node, and the list has only one element, ALOE moves the cursor to that element. In this manner cursor movement is symmetric with respect to lists. If the cursor is at a terminal node it has no effect.

- Cursor **next** and cursor **previous**: Move the program cursor to the next or previous sibling if one exists. If not, they move it to the next or previous sibling of the parent node recursively. One problem with the cursor **next** and cursor **previous** commands is that they are not symmetric. A cursor **next** command followed by a cursor **previous** command does not necessarily move the cursor back to the original position.

- Cursor **home**: Moves the program cursor to the *root* of the *current* window or context. It is equivalent to a series of cursor **out** commands. If the current node is a root itself, it moves the cursor to the root of the previous window in the context window stack (see section 2.6). It, of course, has no effect if applied at the system root. This command lets the user move more rapidly out of contexts.

All these cursor movement commands can take a numerical argument which specifies the number of times that it should be applied. This helps the user move *faster* through the program. This basic set of cursor movement commands can be naturally extended with two other cursor movement commands that would move the cursor following a preorder traversal of the tree. These commands would be symmetric and the user could move the cursor to any

place in his program by using repetitions of the same key. This form of cursor movement has been successfully used in the Synthesizer [Teitelbaum 81a]. In fact, the Synthesizer has thirteen different cursor movement commands. A user must type several cursor movement commands to move from one place to another in his program. The same is true in ALOE, and as long as the system response is fast enough, with practice the user rapidly gets used to typing several cursor movement commands consecutively to get to the desired node.

There are three other important cursor movement commands:

- **.find**: This command lets the user move rapidly to nodes that are not very close to the current node. Very helpful when the desired node is not currently displayed. The .find command restricts the search to the *current* window (see section 2.6) and wraps-around the root to search for nodes that appear before the current node.

  ALOE lets the user specify a search pattern (a string) and moves the cursor to the first node that matches the string. On terminal nodes (constants, variables, metas *etc.*), a substring match is done with the associated value of the node. On non-terminal nodes the match is done with the names of the corresponding operators and their synonyms. So, it is possible to search for variable names, constant values and names of meta nodes as well as for `if` or `while` statements.

- **.back**: Moves the program cursor back to its previous position, provided that no change has been made that invalidates that position. For example, if a node has just been deleted, the previous cursor position no longer exists. The **.back** command is very helpful in situations where the result of another cursor movement command was not the desired one.

- **.window**: This command is used to move from one program window to another, or to move to and from clipped area windows (see section 2.6). Every window has a program cursor associated with it. Applying this command, implicitly indicates a cursor movement to that window's cursor.

The Synthesizer [Teitelbaum 81a] has no searching commands, all cursor movements must be done with one of the explicit cursor movement commands. Even if response from the system is fast, this may not be appropriate for large programs where it would require too many cursor movement commands to get to nodes that are not close to the current node.

In MENTOR [Donzeau-Gouge 80], cursor movement and searching is performed using MENTOL, a tree manipulation language that is the command language of MENTOR. Complex tree pattern matching can be performed in MENTOL to perform searching. In some cases it is very cumbersome to express the pattern to look for, when a simple string match would be

easier to express as it is done in ALOE. For further discussion on the comparison of ALOE with these and other syntax-directed editors see section 6.8.

A pointing device could also be used to indicate cursor movement by moving it to different locations on the screen. Emily [Hansen 71] used a light pen pointing at the screen to indicate cursor movement. In order to do this, a mapping between tree nodes and screen positions must be made. This information is available when the node is unparsed and could be stored with the tree node.

The highlighting of the program cursor and the cursor movement commands strongly emphasize the structure of the program. For new users this is extremely useful in helping them to deal with their programs structurally instead of textually, especially for those that have used text editors.

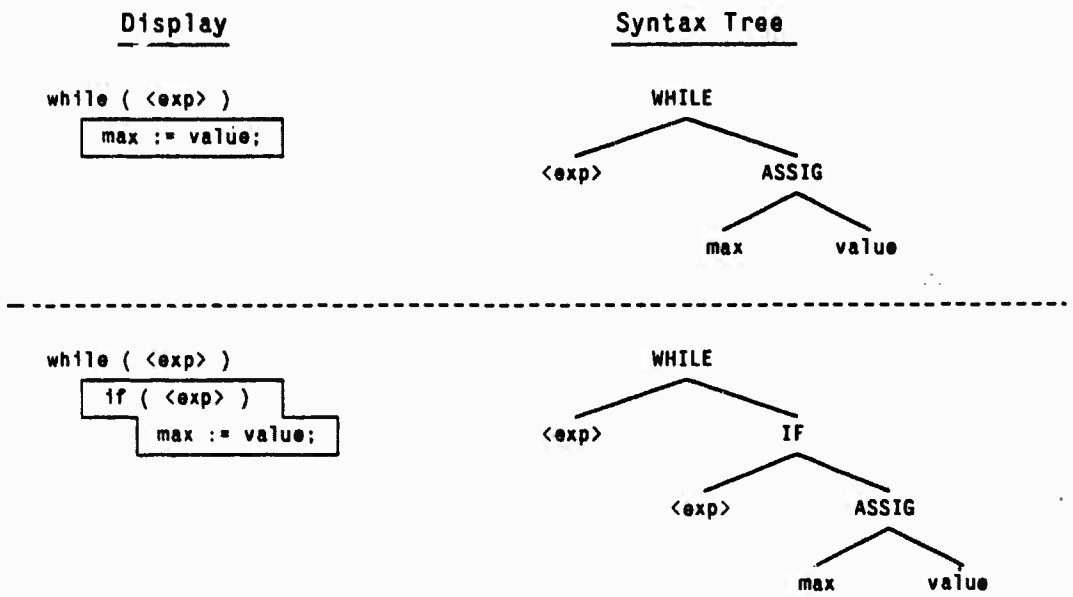

Figure 2-4: Nesting an assignment statement into a if-then statement

2.4.2.2. Tree Manipulation Commands

A set of basic tree manipulation commands is provided as part of the editing commands of an ALOE. These commands are very important because they address the issue of editing structures in syntax-directed editors which has been traditionally considered one of its problems. The basic set of tree manipulation commands include:

- **.clip** and **.insert** used to copy and move subtrees. Clipped subtrees are kept on a separate clipped area where they can be inspected and edited.

- **.delete** and **.replace** used to delete subtrees. The **.replace** command leaves a meta node in the place of the deleted subtree.

- **.nest** and **.transform** are very important because they make editing structures much easier and contribute towards making structured editing much more attractive. The **.nest** command makes a new subtree in the place of the current one with the current subtree as offspring of the new root provided that the resulting subtree is a legal one. Figure 2-4 shows the effect of nesting an `assignment` statement into an `if-then` statement.

  The **.transform** command changes the operator of the current subtree root provided that the transformation is a legal one. Figure 2-5 shows the effect of transforming an `if` statement into a `while` statement. This kind of example is often used to show the difficulties of editing structures in syntax-directed editing because normally it can only be done with a sequence of **.clip**, **.delete** and **.insert** commands [Teitelbaum 81a]. The Synthesizer has been extended to support some transformations [Teitelbaum 82].

2.4.3. Extended Commands

As every ALOE is generated for a different language to form an environment, there is a need for some language or environment specific behavior. One way of achieving this behavior is through the introduction of extended commands to implement language or environment specific operations. The format of extended commands is the same as editing commands: the command name is prefixed with a period (".") and control characters are used as synonyms. It is important to select the names of extended commands so that they do not have common leading characters with the names of editing commands. Their control character synonyms should also be different from those of the editing commands.

Typical uses for extended commands include commands to communicate with other tools of the environment through the uniform user interface of ALOE (*e.g.* communication with the run-time environment through **.run** and **.continue** commands), commands to perform

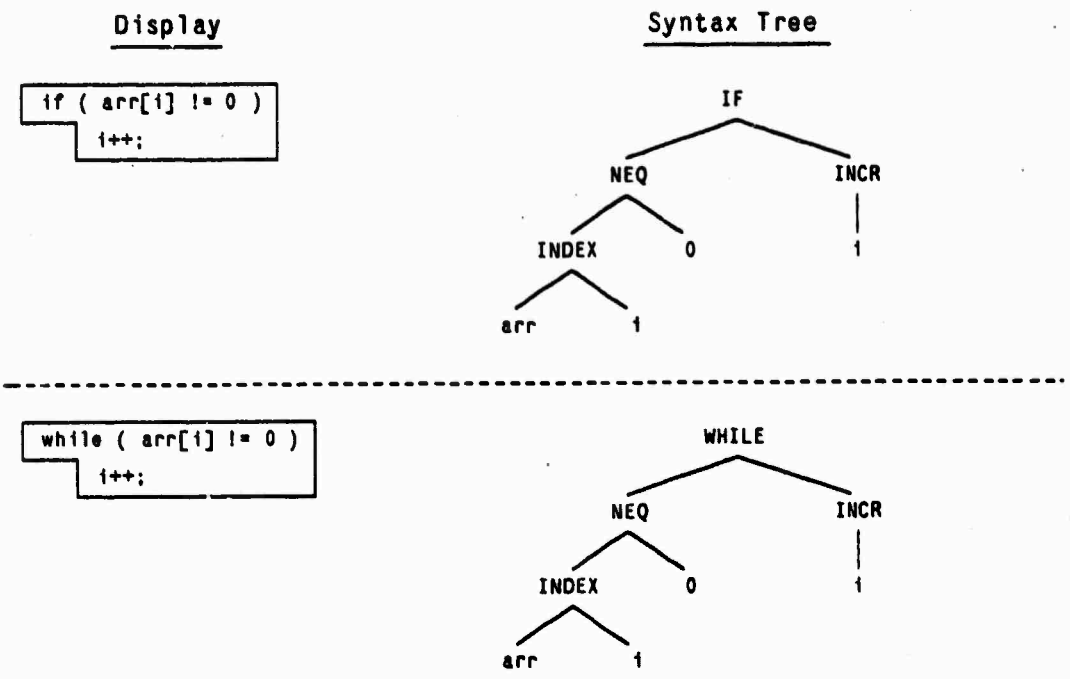

Figure 2-5: Transforming an IF statement into a WHILE statement

language or environment specific searches (*e.g.* a **.find-spec** command to locate the specification of a procedure), *etc.* For further discussion and examples of extended commands see section 5.2.1.

2.5. Multiple Unparsing Schemes

The structure and format of unparsing schemes is discussed in section 3.2.5. In this section the discussion focuses on their impact on the user interface. Unparsing schemes specify the mapping from the internal program representation (defined by the abstract syntax) to the concrete textual representation used for display. In ALOE we can specify several of these mappings for every language operator, thus providing a facility for multiple *views* of the same program. Unparsing schemes are specified by the language designer in the grammatical description of the language.

The ability to specify multiple unparsing schemes is obtained because of the clear separation of the abstract syntax and the concrete representation of the language operators in the grammatical description. If traditional BNF [Backus 59] (or a variant) had been used, this would not have been possible because in BNF the concrete representation *is* part of the syntactic specification.

2.5.1. Different Abstraction Levels

The ALOE implementor can specify different levels of abstraction (or different levels of detail) for the program. The user *sees* these different *views* of the program when the current unparsing scheme is changed. The internal program is the same.

Structured editors for LISP [Teitelman 78], use the concept of *depth* of the tree to present to the user different levels of detail, while *eliding* the rest of the program. MENTOR [Donzeau-Gouge 80] uses the same concept for *elision* or *holophrasting* applied to algebraic languages. The drawback of this design is that program structures that are at the same depth are not necessarily at the same level of abstraction, which then results in many instances in which too much or too little detail is given to the user.

Holophrasting was used in Emily [Hansen 71] as an abbreviation mechanism for a node. A holophrast of a node consisted of the leading portion of its textual expansion together with some distinguishing marks to indicate it. Holophrasting was a property of a node. The Synthesizer [Teitelbaum 81a] achieves a similar effect through the use of comment statements. When the *elision* command is applied to one of such statements, the comment part is displayed in the screen and a set of dots replaces the expansion of the statement. In both cases elision is controlled explicitly on a node-by-node basis, no depth value is used.

The PDE system [Alberga 81, Mikelsons 81] uses a very sophisticated algorithm that assigns *weights* to different nodes in the tree according to their relevance to decide which ones will get unparsed given the size of the screen. The algorithm is evaluated for every redisplay and changes the *weights* for different abstraction levels. This causes an alteration of the display after every cursor movement, which could be very distracting and could cause some delays in the updates.

With multiple unparsing schemes in ALOE, the language designer has the ability to specify

the different abstraction levels. He indicates, for every different scheme, which nodes get unparsed, and how. One good example is the development of unparsing schemes to show a procedure call cross reference in a program. All the information in the program is ignored except for procedure calls. In this manner, the desired abstraction level is achieved.

2.5.2. Pretty Printing and Syntax Variation

Since the unparsing scheme language includes commands that support indentation, column positioning, *etc.*, pretty printing of programs can be done automatically. With multiple unparsing schemes, different pretty printing styles can be specified.

A good example of the use of multiple unparsing schemes can be found in ALOE$_{GC}$. The GC language [Feiler 79] is a variant of C [Kernighan 78] with a different syntax for procedure specification. The abstract syntax of both languages is identical, so ALOE$_{GC}$ has an unparsing scheme to unparse GC programs with C syntax. Figure 2-6 shows a procedure specification unparsed first with GC syntax and then with C syntax.

```
boolean answer(struct fract *fr1, *fr2; char op)

boolean answer(fr1, fr2, op)
        struct fract *fr1, *fr2;
        char op;
```

Figure 2-6: Two different unparsings of a GC procedure specification

Aloegen, an ALOE used to construct an edit the grammatical description, produces a set of tables written in C [Kernighan 78] which constitute the syntax tables used by a generated ALOE. These tables are produced entirely through the use of many different unparsing schemes. More examples of different uses for multiple unparsing schemes can be found in section 5.2.8.

2.5.3. Language Translation

Another example exploits the similarities of GC and PASCAL [Jensen 74]. A large percentage of GC and PASCAL have the same structure, so it is possible to have an ALOE$_{GC}$ with unparsing schemes to show PASCAL syntax for those constructs that have the same

```
struct employee
{
    char name[64];
    int ssn;
    int salary;
};
int maxsalary(struct employee payroll[512]; int num)
{
    int i;
    int max;

    max = -1;
    for (i = 0; i < num; ++i)
        if (payroll.salary[i] > max)
            max = payroll.salary[i];
    return max;
}
```

```
MODULE example;
EXPORTS
    TYPE employee = RECORD
        name : ARRAY [0..64-1] OF CHAR;
        ssn : INTEGER;
        salary : INTEGER;
    END;
    FUNCTION maxsalary(VAR payroll : ARRAY [0..512-1] OF employee;
                       num : INTEGER) : INTEGER;

PRIVATE

    TYPE employee = RECORD
        name : ARRAY [0..64-1] OF CHAR;
        ssn : INTEGER;
        salary : INTEGER;
    END;

    FUNCTION maxsalary(VAR payroll : ARRAY [0..512-1] OF employee;
                       num : INTEGER) : INTEGER;
        VAR i : INTEGER;
        VAR max : INTEGER;
    BEGIN
        max := -1;
        FOR i := 0 TO num-1 DO
            IF payroll.salary[i] > max THEN
                max := payroll.salary[i];
        BEGIN
            maxsalary := max;
            EXIT(maxsalary)
        END
    END;
```

Figure 2-7: A small program unparsed with GC and PASCAL syntax

structure [Feiler 82b]. The equivalent effect can be achieved in an $ALOE_{PASCAL}$. Figure 2-7 shows an example of a small program unparsed first with GC syntax and then with PASCAL syntax. A similar effect could be achieved for common subsets of ADA and PASCAL [Albrecht 80].

2.6. Screen Organization

One of the major limitations imposed by the terminal is the screen size (typically 24 lines by 80 characters, for the kinds of terminals selected). There is a large amount of information that needs to be displayed and not enough screen space is available for it. The screen organization then becomes extremely important for the user interface.

In any ALOE there are many different *kinds* of information that are displayed to the user. It is very important that users always have a clear picture of the system state. This is specially true with new users that will be faced with a totally different system from the ones they are used to. Information of a particular should always appear in the same place in the screen.

The ALOE interface divides the screen in different sections (called *windows*). To take full advantage of the terminal capabilities and to be able to display all the information needed, it is very important to have a sophisticated window interface with different kinds, sizes and functionality of windows. The different kinds of information are then displayed in these windows. In most cases the sizes and distribution of these windows can be modified by the user; however, every ALOE provides a default window layout, which is normally sufficient. Figure 2-8 shows the distribution of the different windows and the different kinds of information in a typical screen layout of an ALOE. Windows limited by dotted lines in the figure are normally only displayed in the screen when a specific request is made for the information that is displayed in them. The clipped and help windows are typical examples of these windows. In simple ALOEs debugging and user I/O windows will never appear. A detailed description of the display and windowing characteristics of an ALOE can be found in [Medina-Mora 81a] and [Feiler 81].

ALOE organizes the different kinds of information as follows:

- Command input. There is a *command* window which is two lines long and is normally placed at the bottom of the screen. The first line of it is used for echoing command input from the user.

- Last command. In the second line of the command window, the full name of the command just invoked by the user is displayed, while ALOE is waiting for a new command to be typed. For novice users it is helpful to show the commands that they applied so that they can learn to associate the commands with their effects on the programs. It is also very useful when a command is typed in error.

- Prompts from the system. Sometimes ALOE needs to prompt the user for values

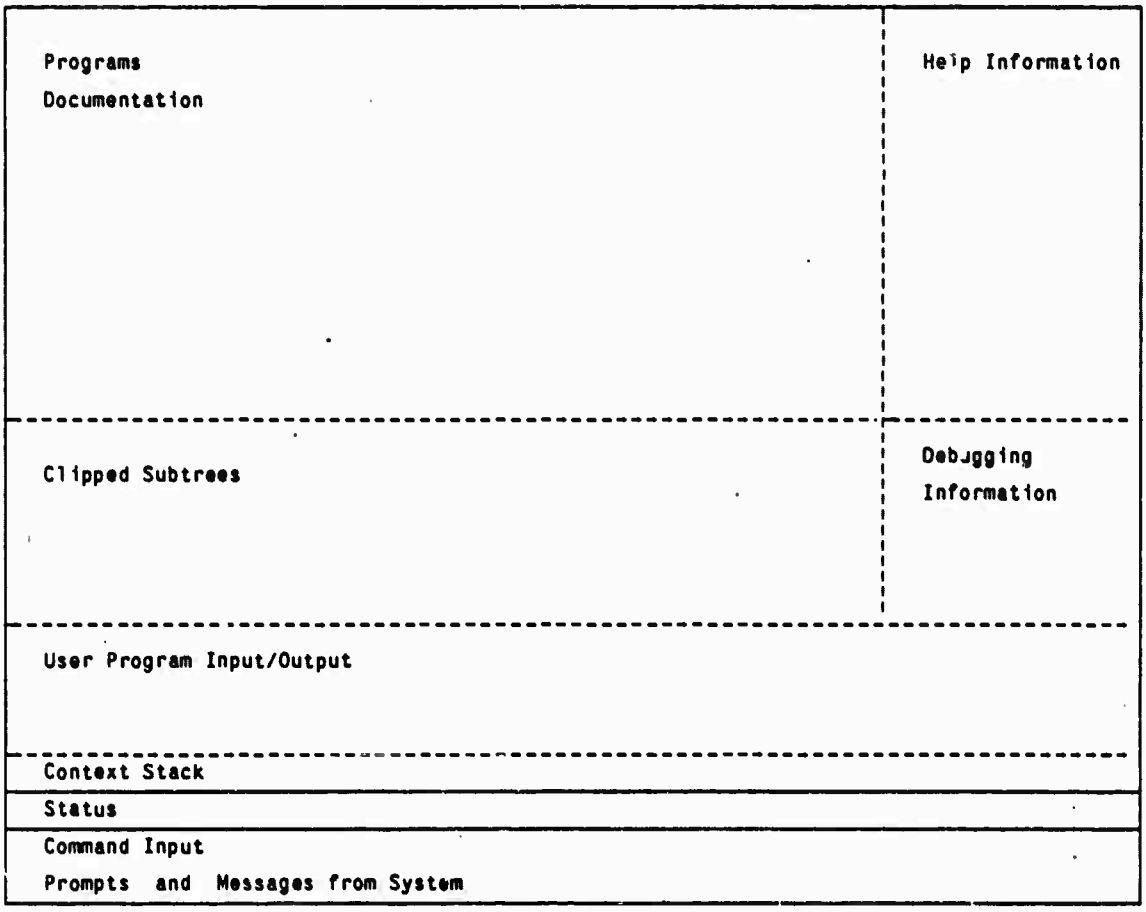

Figure 2-8: Screen Organization in an ALOE

(*e.g.* the name of a variable, the value of a constant, the name of a file, *etc.*). The second line of the command window is used for prompts. The command typed by the user is still in the first line of the command window, so a novice user can understand the need for the prompt.

- Messages from the system. The second line of the command window is also used for different types of messages that the system displays at various times. These include system errors (*e.g.* when an inconsistency is discovered), other system messages (*e.g.* confirmation that a file was successfully written) and messages from action routines and extended commands. The error interface for action routines and extended commands (see section 4.5) combines the display of the message with the highlighting of the node associated with the message.

- Programs. The concrete representation of the internal abstract syntax tree is unparsed into *program* or *tree* windows. ALOE maintains a stack of such windows (referred to as *context stack*). It is useful to divide the program data base into different contexts so that only the current context is displayed at any moment. Previous contexts are stacked and can be accessed through the cursor home

command or the .window command (see section 2.4.2.1). Because of the restricted screen space, every context window is overlaid in the same area of the screen.

As the user attempts a cursor out command at the root of a context window, ALOE makes an action routine call to let the action routine *pop* the context window stack and make ALOE display the previous context. A similar situation occurs when a cursor in command is attempted at a node whose offspring are not unparsed (*i.e.* they are not visible), the action routine can then change the unparsing scheme into one that would unparse the offspring and *push* a new context window into the stack. This can be also visualized as an *opening* of a new context or level of abstraction.

- Clipped subtrees. The **.clip** command is used to copy subtrees from the program tree into a scratch area called the *clipped* area. *Clip* windows are used to display clipped subtrees. The cursor can be moved to a clipped subtree using the .window command (see section 2.4.2.1). The user can then edit the clipped subtree before moving it for insertion elsewhere in the program. The clip window usually appears at the bottom of the program window (sharing part of it) and disappears as soon as the cursor is moved back to the program window. Only one clipped subtree can be displayed at a time.

- Status Information. A one line *status* window is used to display the status of ALOE. This information includes the name of the operator that corresponds to the current node, the class it belongs to, the name of the current context window and the settings of the various modes of ALOE. This status information is extremely helpful to novice users because it helps them understand more rapidly both, the behavior of ALOE and the structure of the corresponding language. The status window is shown in reverse video and is normally placed between the program and command windows, serving also as a convenient window separator between them.

- Context stack window. When different contexts are stacked, it is very useful to see the contents of the stack. The *context stack* window is a one line window that is displayed if there are nested contexts. The context stack window is also shown in reverse video and is normally placed above the status window taking one line away from the program window.

- Help information. The *help* window is used to display the names of all the legal operators and their synonyms when the current node is a meta node. It is also used to show the names of editing and extended commands. The user can request this help information explicitly by typing the **.help** (or **?**) command. It will be automatically displayed when the *novice* mode is set (see section 2.7). When help information is not being displayed, the help window is normally overwritten by the program window.

- Documentation. The program or tree window can also be used to display long pieces of documentation text (defined in the grammar as text constants, see section 3.2.2.1). The document operators can be defined in the grammar to open a new context so that they can be displayed separately from the program.

- Debugging information and other information about the programming environment. An ALOE implementor can also define an additional set of windows for displaying other types of information. In the GANDALF environment, debugging information (see section 5.2.5), such as the monitoring of variables is displayed on a special *monitor* window.

- User Program I/O. If the ALOE is a programming environment in which a user is going to run programs, a separate section in the screen is needed for the input and output of user programs. A *user* window can be allocated for these purposes. The ALOE implementor can then use the windowing capabilities of the ALOE implementation environment (see section 4.6) to provide the user program with I/O capabilities. Normally the user window will use a small part of the screen because it is desired of it to coexist with the program window that displays the user program that is generating the output or requesting the input.

After every interaction with ALOE the screen is updated to reflect the new state. If the communication line baud rate would be large enough to allow instant redisplay, the ALOE would simply redisplay the screen image every time. Since this is not the case in general, it is important to have an intelligent display interface that would update the screen as fast as possible using the terminal capabilities to update only the parts of the screen that have changed. To this end, ALOE uses the display package developed for UNIX[tm] EMACS [Gosling 81a, Gosling 81b]. The package uses an optimal algorithm for redisplaying the screen every time it changes.

Better displays could clearly be exploited to provide an improved screen organization. The current system has been developed for a Concept-100 terminal [HDS 79]. Displays with similar characteristics could also be used with similar results.

One of the first improvements would be to take advantage of terminals with larger screens or with additional local memory. Several screen images could be stored in the terminal's memory, so switching from one screen image to another would be instantaneous. This capability could be used to help solve the problems, discussed above, related to switching between different displayed contexts, to the coexistence with user program I/O as well as to provide a larger window size for several of the above mentioned windows.

Beyond this, terminals with raster scan displays are more suitable for windowing and distribution of different kinds of information [Teitelman 77, Sproull 79, Ball 80, Ball 81]. The use of these kinds of displays would greatly improve the user interface (and the usability) of ALOEs.

With respect to user input to ALOEs, the current system only supports keyboard input. Although the use of control characters and function keys has greatly improved the user interface, a pointing device, such as a *mouse*, coupled with the display's improvements discussed above would make possible the use of menus that have been used successfully in ...r interactive systems [Teitelman 77, Lampson 79, Ingalls 78].

2.7. Modes of Operation and User Profiling

One goal of interactive systems is to have enough flexibility to be able to deal efficiently with the different levels of expertise of their users. Novice users should find it easy to learn how to use the system. Experienced users should be able to achieve what they want to do without an excessive effort demanded from them.

For these purposes every ALOE operates in two different modes: *novice* and *expert* mode. In novice mode a help window is constantly maintained and is used to display the list of language commands available to the user. After every command is given the help list gets updated accordingly. In expert mode no continuous help is maintained, but the user can ask ALOE to display the help window whenever it is needed, by typing the **.help** command. A user profile could be used to provide the default value for this mode.

Most languages will have unparsing schemes that specify that nodes be unparsed in the same order as they appear in the abstract syntax specification. However, there are some languages for which the user would like to build programs *from left to right* (e.g. functional programming languages such as ALF/. [Habermann 80]). To achieve this, the unparsing schemes specify a different order of unparsing than the abstract syntax tree, and the ALOE would follow the order of the tree while moving the cursor from one node to the next, thus achieving the desired effect. Figure 2-9 shows the effect of building a function composition in $ALOE_{ALFA}$. The components of the operator COMP are unparsed in reverse order, and appear then to be constructed from left to right.

In the case of searches however, one user might want ALOE to follow the internal tree order while another user might want it to follow the textual order which is the one that the unparsing schemes define. To provide the needed flexibility to achieve the desired behavior every time, ALOE defines a mode that affects the tree traversal order. On the normal setting the program cursor follows the internal tree structure. It can be changed so that the program cursor

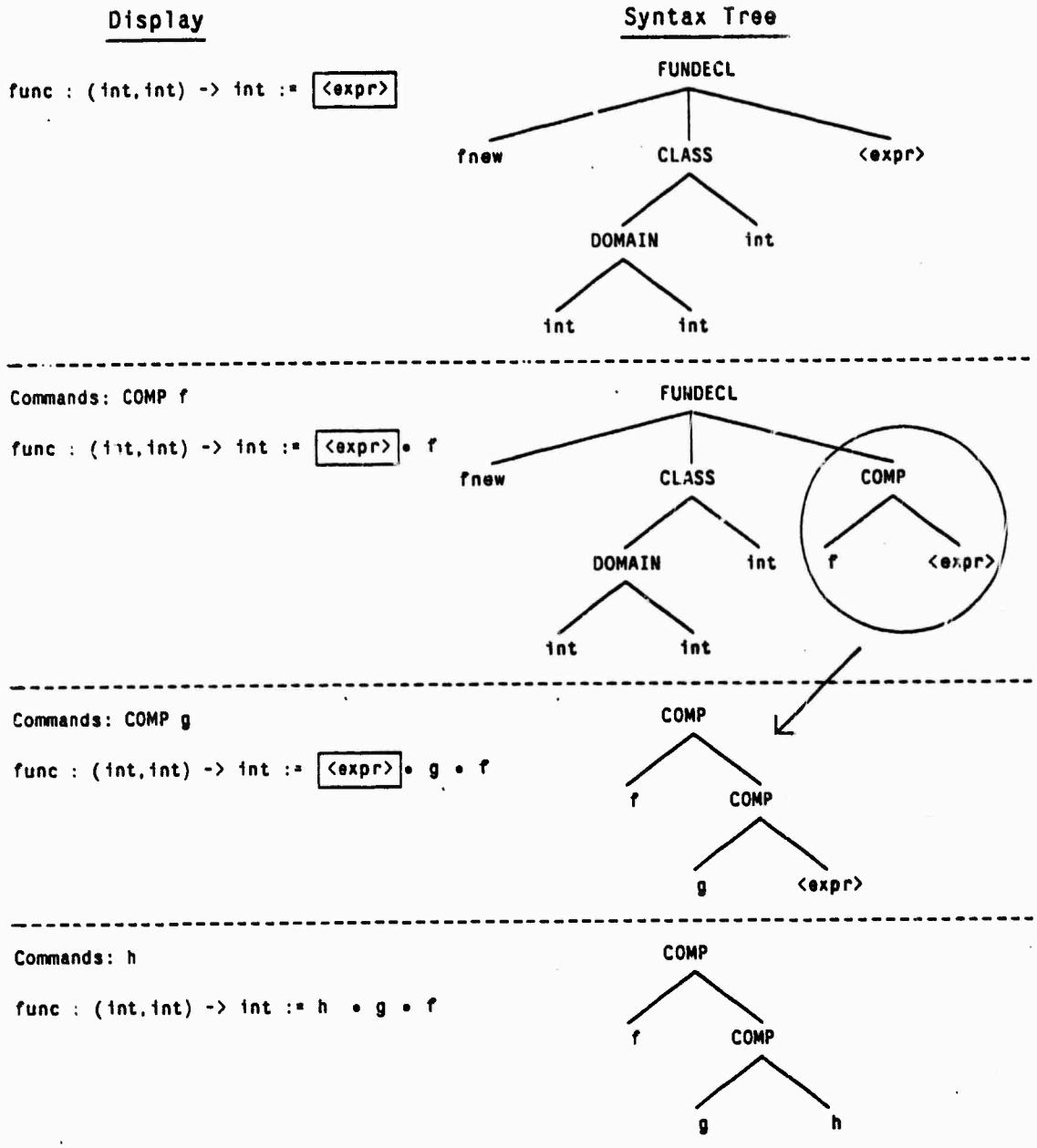

Figure 2-9: Building a Function Composition in Alfa

follows the concrete textual order. In either setting the program cursor will not be moved into a node that is not currently unparsed, *i.e.* if it is not referenced in the current unparsing scheme.

The need for this mode was not anticipated because it was thought that all languages would unparse their internal structure into text in the same order. It was learned that this was not the case when ALOE$_{ALFA}$ was generated. For other languages that do not have this kinds of constructs, the mode setting is not important.

As part of the constructive approach to program building in an ALOE, the program cursor is moved to the next meta node after every constructive command. The space in which the search for the meta node is performed varies according to the setting of the *editmode*. When in *constructing* mode, ALOE searches the visible context, when in *editing* mode the search is restricted to the particular subtree where the previous operation took place. This avoids the undesirable effect of having the cursor moved to an unrelated section of the program that happens to have a meta node left. The **.find** command can be used to search for any meta nodes left.

For ALOE environments that use the facilities provided by the stack of context windows the setting of the *show-windows* mode causes the context stack window to be displayed. This mode is only meaningful for ALOEs that will have different contexts defined.

The window layout on the screen can also be defined through a user profile. ALOE reads a file that contains the definition of the window layout and every user can then define it to his convenience. This profiling mechanism could be extended to provide users with the ability to choose between different formatting styles and other such features that could be customizable in an ALOE.

Chapter 3

The ALOE Generator

3.1. Introduction

In this chapter we will discuss the issues involved in the generation of syntax-directed editors, as well as the structure and properties of the ALOE generator. The ALOE generator produces an editing environment based on a grammatical description of a language. A wide variety of editing environments can be generated: from simple stand-alone editors with only syntactic knowledge to very complex software development environments such as GANDALF [Habermann 82], discussed in chapter 5. In section 6.6 we will discuss how the design decisions are influenced by the differences of generating syntax-directed editors as opposed to building a hand-crafted editor for a particular language.

Every generic system like the ALOE generator will have a kernel, common to all the generated systems. It also needs to have some form of input that specifies the language or structure for which an editor is going to be generated. If the resulting product is to have some language specific behavior, some extensions should be provided. Consequently, every ALOE generated has three major components:

- The ALOE kernel, common to all ALOEs. It understands and manipulates the internal representation of trees (*e.g.* programs) and is language independent. It provides an extensive set of editing commands for tree manipulation, cursor movement, input/output, display manipulation, *etc.* (see chapter 2). It also provides the default implementation of the set of environment specific functions such as the symbol table manipulation (see chapter 5 for a detailed discussion of these functions).

- The syntactic tables produced from the grammatical description of the language. They provide ALOE's syntactic knowledge of the language, as well as the unparsing knowledge through the unparsing schemes.

- Implementation of action routines. extended commands and environment specific routines. These provide the language specific behavior of an ALOE that is beyond its syntactic capabilities. Action routines are discussed in detail in chapter 4. The use of extended commands in a large integrated environment is discussed in 5.

3.2. The Grammatical Description

The most important contributions of the design of the grammatical description are:

- Clear separation of abstract syntax and concrete representation of the operators of the language (discussed previously in chapter 2). The ALOEs directly manipulate the abstract syntax; all interactions are in terms of this syntax. The concrete representation specification is only used for display purposes. This allows the implementor of an ALOE to define multiple concrete representations for a single abstract syntax.

- The grammatical description defines an important part of the user interface. The names of the language operators are the names of the constructive or language commands of the ALOE (see chapter 2).

The grammatical description follows certain syntactic rules and its syntax can be expressed in terms of itself. This means that an ALOE can be generated to create and manipulate grammars. This ALOE will be referred to as *aloegen*. The concrete representation of grammars is no longer important just as concrete representations of other languages are no longer important in the ALOE context.

The following sections discuss in detail each one of the conceptual pieces that form the grammatical description. These pieces are interleaved in the actual description. To facilitate the discussion, this is a list of the conceptual pieces:

- The name of the language.

- The abstract syntax description of the language.

- The root operator.

- The precedence of non-terminal operators.

- The unparsing schemes that describe how the (internal) tree form is displayed on the screen. They describe (potentially many) mappings from the abstract syntax to the concrete representation.

- The action routines associated with each operator of the language, needed to expand on the syntactic capabilities of the basic ALOE.

- The synonyms available for each operator.
- Whether or not the corresponding node of a non-terminal operator should be a root of a separately stored tree.

Figure 3-1 shows the grammatical description of a very simple language that implements a small interpreter. Throughout this chapter, this example will be used to illustrate the different parts of the grammatical description. The grammatical description used in MENTOR [Donzeau-Gouge 80] is similar to the one described here, in section 3.2.9 we will discuss their differences.

3.2.1. The Name of the Language

The name of the language is specified as part of the grammatical description of that language. It is really a name of the ALOE version for a language. This name is encoded in every tree that an ALOE writes and is used to prevent an ALOE for one language from editing a tree of a different language or a tree of a different version of the same language. In the example of figure 3-1, the name of the language is INTERP.

During development of a grammar for a new language, changes are made to the grammar that make the trees created by an earlier ALOE version no longer manipulable by the new version. The ALOE implementor does not always change the name of the version whenever there is a change in the grammar, and inconsistencies may develop when the ALOE tries to manipulate a tree that is really of a different version.

This situation can be dealt with through action routines of *aloegen* (the ALOE for grammars), that could change the name of the version whenever a structural change is made to the grammar. Since only the abstract syntax structure is stored in the files and changes to the unparsing schemes do not affect this structure, it is consequently not necessary to change the version name after making modifications to the unparsing schemes.

There are certain changes that should not cause incompatibilities in the files. For instance, adding new operators should not make previously stored trees that do not use these operators, incompatible. The problem is that the format of the tree files includes references to the table of operators that defines the language. These references change when new operators are added unless the operators are added at the end of the list of operators. They also change when operators are deleted.

```
Language Name: INTERP
Root Operator: PROGRAM
{       /* terminal operators */
LOOPVAR =       {v}
              | (0) "@s"
              | action: <none>
              | synonym: "'" ;
INT =           {c}
              | (0) "@c"
              | action: aINT
              | synonym: "#" ;
EMPTYSTEP =     {s}
              | (0) "1"
              | action: <none>
              | synonym: <none> ;
}
{       /* non-terminal operators */
PROGRAM =       stmts
              | (0) "@1"
              | action: <none>
              | synonym: <none>
              | precedence: <none>
              | Filenode;
PRINT =         <exp>
              | (0) "print @0,"
              | action: <none>
              | synonym: <none>
              | precedence: <none>
              | Non-filenode;
FOR =           loopvar exp exp stepexp stmts
              | (0) "for @1 = @2 to @3 step @40+@n@50-"
                (1) "for (@1 = @2; %1 <= @3; %1 =+ @4)@+@n@50-"
              | action: <none>
              | synonym: <none>
              | precedence: <none>
              | Non-filenode;
PLUS =          exp exp
              | (0) "@1 + @2"
              | action: <none>
              | synonym: "+"
              | precedence: 1
              | Non-filenode;
TIMES =         exp exp
              | (0) "@1 * @2"
              | action: <none>
              | synonym: "*"
              | precedence: 2
              | Non-filenode;
STMTS =         <stmt>
              | (0) "@0@n"
              | action: <none>
              | synonym: <none>
              | precedence: <none>
              | Non-filenode;
}
{       /* classes */
stmts           =       STMTS ;
exp             =       INT LOOPVAR PLUS TIMES ;
loopvar         =       LOOPVAR ;
stepexp         =       INT PLUS TIMES EMPTYSTEP ;
stmt            =       PRINT FOR ;
}
```

Figure 3-1: Grammatical Description of a Simple Language

This problem could be avoided through action routines of *aloegen* which could generate a new reference every time a new operator is added, and not reuse the reference for a deleted operator, although this could ultimately result in a very large table with many empty entries. This does not solve the problem of modifying the abstract syntax description of a particular operator: adding a new offspring, deleting one or changing the order of them.

What is needed here is a general mechanism to map one tree into another given the changes to the grammar. This is not necessarily an easy task and is probably not worth providing it, if one considers that during development of a new grammar all programs are *illegal* until the final form of the grammar is developed. There may be some exceptions to this situation. The grammatical description itself is one: sometimes it is desired to change it to add a new piece of information (for example, when synonyms were added). These changes could cause previous tree grammar files to be incompatible. Other possible exceptions are grammars for complex language systems, such as GANDALF [Habermann 82], in which new features may be added through modifications to the language.

Other systems that deal with structured internal representations must deal with this problem too. In the IDL Translator [Lamb 82] an attempt is made to provide the mechanisms to perform the transformations on the structures when changes are made.

This problem does not arise in parser generators because the programs are kept in textual form and are always reparsed into the internal representation. Changing the grammar for the parser generator does not make the old program incompatible. It may cause it to have syntactic errors according to the new grammar but it still can be processed. The important difference here is that no implicit language information is encoded in the text file.

3.2.2. Abstract Syntax Description

The description of the syntactically correct sentences in the language is done in a two level structure. One level provides the description of each language construct (called operators of the language). There is one production in the grammatical description for every operator. Operators correspond directly to nodes in the internal tree. There are two possible descriptions of this type: terminal operators, whose corresponding nodes represent the leaves of the tree (*e.g.* an integer constant or an identifier) and non-terminal operators, whose corresponding nodes represent either an ordered set of offspring or a list of offspring (*e.g.* an

IF statement or a list of variables). The second level of the description lists the classes of the language. Classes represent the set of legal operators that can be created in a place of an unexpanded offspring (referred to as a *meta* node). In the example of figure 3-1, the class exp includes the operators INT, LOOPVAR, PLUS and TIMES.

A good way to look at the grammatical description is that the sentences form AND/OR structures [Tichy 82] where the operators provide the AND functions (the structure of the offspring is uniquely determined) and the classes provide the OR functions (a choice from a set of operators is specified).

3.2.2.1. Terminal Operators

There are several types of terminal operators. They differ from each other and from non-terminal operators with respect to user input interface (some constant types have embedded blanks, others do not), and with respect to their internal node representation. The internal representation is not important to an ALOE user but it is important to an ALOE implementor who must write action routines. For a detailed description of the internal representation see [Medina-Mora 81a].

The production for a terminal operator in the grammar specifies its type, the set of unparsing schemes, the action routine name and the synonym for the operator. In the example of figure 3-1 the terminal operator INT is specified to be of type *constant*, with one unparsing scheme, an action routine called aINT and '#' as its synonym.

The original design included the following set of terminal operator types:

- *Static* operators are used for some concrete pieces of the language like the name of a type (*e.g. float, integer, boolean, etc.*). The corresponding nodes contain no other information.

- *Constant* operators are used for integer or character constants. Their corresponding nodes contain ASCII values other than blanks.

- *Variable* operators are used for the variables of the language. Names of variables are entered into a symbol table. The ALOE kernel provides an interface to a very simple name table structure which can be replaced by a more complicated symbol table mechanism (see sections 3.3 and 5.2.4).

As more grammars that included string constants and comments were developed, it became apparent that constants with embedded blanks should be treated differently.

Experience with long comments and the creation of the GANDALF grammatical description [Notkin 82b] (see chapter 5), which included documentation and log messages, underlined the need for an underlying text editor to deal with long pieces of text. Two new terminal operator types were then introduced:

- *Long Constant* operators are constants whose corresponding nodes contain ASCII values including blanks. These operators are used for language constructs such as comments and string constants. Their internal representation is identical to constant operators. They differ with respect to user input interface: the input is different because of embedded blanks.

- *Text Constant* operators are used for long pieces of text that require text editing capabilities for their creation and modification. Again, they differ from the other constant operator types with respect to user input interface: an external text editor (UNIX EMACS [Gosling 81a]) is used for creation and modification of text constants.

The additions of long constants and text constants were motivated more by issues of user interface than by issues of lexical analysis. But one might conceive of a larger set of constant operator types to deal with lexical differences. For example, we could have integer, real, character constants. ALOE could perform automatic lexical analysis to validate such constants as they are entered. However, the lexical rules can change from language to language. What is needed is a specification of the lexical rules as part of the grammatical description.

In the current ALOEs, lexical analysis (if desired) is performed by the action routines associated with the constant and variable operators. Lexical analysis is only needed for constants and variables and it differs from the lexical analysis needed in a compiler where keywords, brackets, punctuation marks, *etc.*, must be identified.

Text editing capabilities should be incorporated into ALOEs to be able to edit the textual pieces in the context where they appear, rather than editing them in a separate environment. It is desirable to be able to implement this via an interface to an existing text editor rather than having to write a new one.

Unfortunately most of the good existing display editors have their own idea of the screen and window manipulation so that they can take *reasonable* advantage of it. In the ALOE context the text editor would have to interact with ALOE's handling of the display and its windowing capabilities as well as with the unparsing process. To edit textual pieces in the

context where they appear means to have the surrounding pieces of program displayed at the same time, which is what the Synthesizer [Teitelbaum 81a] does. The difficulties in this case are with respect to the growing or shrinking of the size of the textual piece being edited. Another source of difficulty is the operating system interface. Different editors require different "settings" of the terminal I/O handler. A new implementation based on ALOE's display and unparsing mechanisms may be easier to do. This may be definitely worth doing in the context of generating an ALOE for a document production system like SCRIBE [Reid 80] whose terminal operators would be paragraphs of text.

3.2.2.2. Non-Terminal Operators

There are two types of non-terminal operators: fixed arity operators and variable arity (*i.e.* list) operators. Fixed arity operators are described by listing the (ordered) set of offspring. Each offspring is represented by a class that specifies the set of legal operators for it. In the example of figure 3-1 the FOR operator has five offspring of different classes.

Variable arity operators are described by specifying the name of the class from which elements of the list must be selected. In the example of figure 3-1 the STMTS operator has a variable number of offspring of class `stmt`. Lists in an ALOE can be empty, that is, they are defined to be lists of zero or more elements.

The production for a non-terminal operator in the grammatical description specifies its abstract syntax structure (*i.e.* if the operator is of fixed or variable arity and the classes of the offspring); the set of unparsing schemes that specify the different mappings from the abstract syntax into the concrete representations; the name of the action routine associated with the operator; the precedence value of the operator; a synonym for the operator; and whether or not the corresponding node should be a root of a separately stored subtree file. All the information after the unparsing schemes is optional. In the example of figure 3-1 the FOR operator has five offspring of different classes, two unparsing schemes and no action routine, precedence or synonym specified.

3.2.2.3. Classes

Classes provide the OR functions of the AND/OR structure that defines the abstract syntax of the language in the grammatical description. The class represents the set of legal operators that can be created to replace a meta node of a particular offspring of a non-

terminal node. Meta nodes are automatically given the name of the corresponding class. It is then clear in the display the kind of operator that is expected.

There are some cases in which a class will have only a single non-terminal operator. For example, if an implementor wants to have the effect of a list of pairs, in the grammatical description he would have to specify an extra level for the pair. This extra level is a fixed arity node with two offspring. The class of elements of the list then contains this non-terminal operator as its only member. Another case arises when the desired effect is to have one of the offspring of a fixed arity node be a list: an extra variable arity operator must be introduced as the offspring.

3.2.3. The Root Operator

This is simply the *start* or *distinguished* symbol of the grammar. The root operator is invoked as a constructive command on startup of an ALOE. The resulting node is referred to as the *system root*. In the example of figure 3-1, the operator PROGRAM is the root operator.

3.2.4. Precedence

Non-terminal fixed arity operators may have a precedence value associated with them. These values are used to provide automatic parenthesization while unparsing into infix notation. They are mostly used for expressions which are the constructs of the language traditionally shown in infix notation. In the example of figure 3-1 the operator TIMES has a precedence value of 2 and the operator PLUS has a precedence value of 1.

For the correct parenthesization of expressions it is also necessary to know if the operator is associative in addition to its precedence. For example 'a - b - c' is not equivalent to 'a - (b - c)' because the operator "-" is not associative. If associativity is not handled, expressions like this must be broken into separate expressions, if the textual form of a program is to be used.

It should be clear, however, that precedence and associativity values are only needed for the concrete representation mapping if expressions are going to be unparsed in infix notation. They are not necessary for the correctness of the internal representation: the abstract syntax description is unambiguous. In fully developed programming environments, such as LOIPE

[Feiler 82a] and GANDALF [Habermann 82], code is generated from the internal tree and thus the correct code is always generated regardless of possible ambiguities in the concrete representation.

ALOEs are designed so that the *valid* version of the structures or programs created is the internal tree representation. The different concrete mappings should be only for display and readability purposes. Nevertheless we have tried to include all the necessary mechanisms so that every ALOE generated can produce a correct and unambiguous textual representation of its structures or programs (at least for the main unparsing scheme). After all, the concrete representation is the only feedback the user has about his structure. Precedence and associativity values are one such mechanism. Section 6.4 discusses more issues about ambiguity of language constructs. Section 6.2.2 discusses more issues about unparsing expressions.

3.2.5. Unparsing Schemes

The clear separation of the abstract syntax from the concrete representations of structures or programs allows the ALOE implementor to define multiple concrete representation mappings for a single abstract syntax. This is one of the major advantages of the design of the grammatical description. Sections 2.5 and 5.2.8 contain a collection of examples and applications of multiple unparsing schemes. Using the windowing capabilities and the action routine interface the language developer can define many different *views* of the system by having new windows defined at different levels of the grammar and using multiple unparsing schemes.

Unparsing schemes are used only as a means to generate a textual representation from the *valid* internal tree representation for display and readability purposes. Unparsing schemes consist of running text mixed with formatting commands. Some of the available unparsing commands include:

- Commands to identify the type of node (like constant *vs* variable), so that its representation can be correctly retrieved.
- Commands to deal with indentation, spacing and line breaks.
- For non-terminals the commands define recursive invocations of the unparser to process the offspring. For fixed arity operators the order of unparsing can be different than the internal order.

- For lists the scheme specifies the separator between elements of the list and a special unparsing to be used when a list is empty, can be specified.

- Commands to dynamically change the unparsing scheme used.

For a complete set of the unparsing commands see appendix B In the example of figure 3-1, the operator FOR has two unparsing schemes which would produce the following two different representations:

```
for i = 4 to 8 step 2
    print (i + 3) * i, i

for (i = 4; i <= 8; i =+ 2)
    print (i + 3) * i, i
```

Unparsing schemes are also used to specify cursor movement (see section 2.4.2.1) depending on the value of a mode. The mode specifies whether the internal representation ordering of the nodes (the one defined by the abstract syntax structure) or the unparsing order determines the order of cursor motion. This can be used to specify right to left movement in some structures instead of left to right by unparsing its offspring in inverse order.

Unparsing schemes also define the *visibility* of nodes in the tree. There are three different possibilities: either the node is not visible at all (it is not referenced in the unparsing scheme), or it is visible but not a legal cursor position (it will be unparsed but the cursor will skip it), or the node is both displayed and a legal cursor position. The second possibility can be used when structures have a name that is repeated in several places but it is desired that the user only edits it in one place with the other places reflecting the changes immediately, or to disable the modification of a particular node, by not letting the cursor stop there, while still showing it on the display. Section 6.5.3 contains an evaluation of the use and limitations of multiple unparsing schemes.

3.2.6. Action Routines

One action routine may be associated with every language operator. These action routines will be called by ALOE in a variety of situations, such as the creation of a node, deletion, cursor movement, *etc*. Instead of one routine per operator, an alternate design could have a set of action routines specified in the grammar for each operator: one for each kind of action call.

The former was chosen because it keeps the grammar much simpler and because the action routine implementor can then decide if he wants to have a separate routine for each action or if he wants to share some code for several of the calls. In some cases it is desirable that several calls have similar behaviors.

These action routines add language specific functionality to an ALOE that is usually non-syntactic. They can be used for a wide variety of tasks from semantic checking to manipulation of the display. Chapter 4 discusses in detail the action routines interface and their applications. In the example of figure 3-1 the INT operator has aINT as its action routine.

3.2.7. Synonyms

As described above, the grammar defines part of the user interface when it defines the names of language operators. This is fine for high level structures especially because the editor needs only as many characters as needed to specify a name unambiguously. However, at the expression level it would be a bad user interface if the user had to specify PLUS or even P for an addition operator. To solve this problem synonyms for operator names were introduced and are specified (optionally) in the grammar. In the example of figure 3-1, a '+' is defined as the synonym of the PLUS operator, and '*' is defined as the synonym of TIMES. The user can invoke the constructive command using the operator name or its synonym.

It is also the case that the operator could have been called '+' instead of PLUS avoiding the need for a synonym in this case. There are other cases like the assignment operator in C [Kernighan 78], in which if the implementor were to use '=' as the name of the operator instead of ASSIGNMENT, with '=' as the synonym, a user not familiar with the language would have no way of knowing that this was an assignment operator instead of an equality operator. In other situations there can be different types of similar operators that could have the same synonym. For example, different types of identifiers could have different operator names (*MIDENT* for module identifier, *FIDENT* for function identifier, *etc.*), with the same synonym as long as they do not appear in the same class. This way the user can uniformly use the synonym for all the identifiers, while the ALOE implementor can still be able to differentiate between these operators for unparsing purposes or for differences in the action routine implementations.

3.2.8. File Nodes

The final part of the description indicates whether or not a non-terminal operator has associated with it a *file node*. This indicates that the non-terminal is the root of a subtree that is stored in a separate file (for storage and checkpointing purposes). The root operator (see section 3.2.3), the operator PROGRAM in the example of figure 3-1, is automatically set by *aloegen* to have a file node associated with it.

File nodes are used to achieve the separation of programs and data bases into smaller pieces. They have information about the file that contains the subtree. File nodes can also have a symbol table associated with them, in which case the name of the file where the symbol table is stored is kept in the file node. Unparsing schemes can be used to *hide* the subtrees that are pointed to by the file node: a special unparsing command in the unparsing scheme of the non-terminal indicates whether or not the subtree is *visible* and should be unparsed. Checkpointing and visibility rules can be coupled, but they are two different mechanisms that can be used separately.

3.2.9. Comparison with Other Grammatical Descriptions

Other grammatical descriptions and formalisms, such as BNF [Backus 59] and the MENTOR grammar [Donzeau-Gouge 80], have been used for syntax-directed editors. One of the early efforts in building an editor generator using a grammatical formalism was the Emily System [Hansen 71]. It used a modified BNF as its formalism. In automatically generated syntax-directed editors the grammatical formalism defines part of the user interface and thus the formalism should be designed taking this into account. In BNF and other formalisms, productions are grouped together forming hierarchies of productions. For any particular construct of the language there is a hierarchy of productions leading into it. Example 3-2 shows a typical BNF description for expressions.

Precedence of operators is handled through this hierarchy concept. As we can see, `arithmetic expressions` have `simple arithmetic expressions`, which have `terms`, which have `factors`, which have `primaries` as offspring.

The Emily system [Hansen 71] takes the first step towards solving this hierarchical problem at the expression level by having all non-terminal operators at the same level. Terminal

```
<arith-axpr> :=          <simple arith-axpr>
                       | <if clause> <simple arith-expr>
                             else <arith-expr>
<simple arith-axpr> :=    <term>
                       |  <adding operator> <term>
                       |  <simple arith-axpr>
                             <adding operator> <term>
<term> :=                 <factor>
                       |  <term> <multiplying operator> <factor>
<factor> :=               <primary>
                       |  <factor> ↑ <primary>
<primary> :=              <unsigned number>
                       |  <variable>
                       |  <function designator>
                       |  (<arith-expr>)
<adding operator> :=                +
                                 |  -
<multiplying operator> :=           x
                                 |  /
```

Figure 3-2: A typical BNF description for expressions

operators are still at a different level in the hierarchy. It is necessary to go through an extra level for identifiers when constructing simple expressions such as 'a + b'. BNF is a very good formalism to be used for parser generators and other such systems but does not provide a good user interface. Aside from the problem with hierarchies, the concrete representation of the constructs is mixed with the abstract syntax, as we can see in the example 3-2. In Emily, the productions that could be applied at any particular point were displayed in a menu. By mixing the concrete representation in the production, the space available for the menu is filled up very quickly, and in this case, the user has to request to see more of the menu. This just makes it confusing for a user when he tries to choose which production to apply.

The ALOE generator grammatical description solves the problem by providing a *flat* structure instead of a hierarchical one, and by separating the abstract syntax specification from the concrete representation. All expression operators are at the same level (they are members of the same class). Precedence and associativity values are used to provide the correct parenthesization of expressions. Parser generators, such as YACC [Johnson 75] also use precedence and associativity values

MENTOR [Donzeau-Gouge 80] uses a similar grammatical description as that of ALOE. It distinguishes between operators of the language with abstract syntax specifications, and classes to group them. It does not have multiple unparsing schemes, specification of action routines, precedence values, synonyms or file nodes.

3.3. The ALOE Kernel

As described above, the kernel is common to all ALOEs. It provides an extensive set of editing commands that include, among others, commands for insertion, deletion, clipping, cursor movement, searching, scrolling, reading and writing from files as well as more complicated tree manipulations such as nest and transform (see section 2.4.2). It also provides the language independent command interpreter that distinguishes between editing and constructive commands, as well as the table driven constructor that lets the user replace meta nodes with legal operators only (the tables are generated by *aloegen* as an alternate unparsing of the grammatical description). Finally, the kernel provides default implementations of the environment specific routines described in the next section.

3.4. Extensibility

ALOEs can be extended to have language or environment specific behavior. There are three different kinds of mechanisms to provide these extensions: implementation of action routines, extended commands and environment specific routines.

Action routines are used to implement language specific functionality such as programming language semantic checking, access control, automatic generation of program pieces, interface to other parts of the environment such as code generation and debugging, *etc.* Chapter 4 discusses these applications in detail.

Extended commands are added to the basic set of editing commands to expand on the capabilities of an ALOE. Some examples of possible extended commands include commands for running programs, continue execution after interruption, environment specific searches, *etc.* Chapter 5 discusses some of the applications of extended commands for a large integrated software development environment.

Experience has shown that some ALOEs desire certain functionality that can not be produced simply through action routines or extended commands. A clear example of this is the symbol table manipulation. Different environments may want to implement different symbol table mechanisms. The kernel provides a uniform interface to the symbol table mechanism through a set of routines. A default implementation of these routines is provided for ALOEs in which a simple name table mechanism is sufficient. For others, some or all these

routines can be replaced (sometimes only a couple of them need to be replaced) by the routines that implement more complex symbol table schemes. A full description and specification of these routines is given in the ALOE Users' and Implementors' Guide [Medina-Mora 81a].

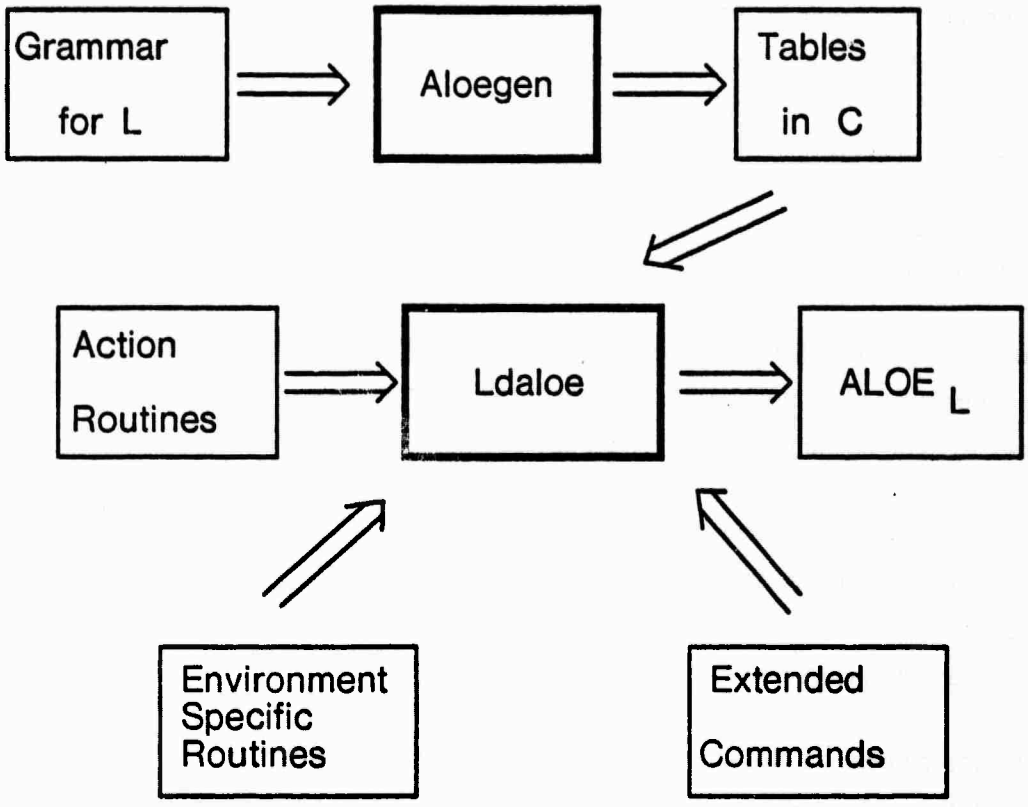

Figure 3-3: The ALOE Generation Process

3.5. The Generation Process

The ALOE implementor goes through a set of simple steps in order to generate an ALOE. First, the grammar is created or modified with *aloegen*, the ALOE for grammars, which produces the language tables and compiles them [Notkin 82c]. Then, the implementations of action routines, extended commands and environment specific routines are written and compiled. Finally, *ldaloe* is invoked which will load the language tables together with the

language specific implementations to produce an ALOE. Figure 3-3 illustrates the ALOE generation process.

This process has been successfully applied on a wide variety of languages:

- A set of algol-like languages such as GC (GANDALF-C [Feiler 79], a slight variant of C) which includes a parser for programs constructed outside of ALOE, C [Kernighan 78], PASCAL [Jensen 74], ADA [DoD 80], different subsets of C and Ada, *etc*.

- GANDALF, a large integrated software development environment whose language is a combination of a language for system version control [Kaiser 82], project management [Habermann 79a], and the programming language (in this case GC). Chapter 5 elaborates on the issues that arise in the implementation of such a large environment.

- A functional programming language, Alfa [Habermann 80] through which we have gained experience with ALOEs for non algol-like languages.

- *Aloegen*, an ALOE for the grammatical description. One unparsing scheme shows the grammar syntax, the others generate the language tables in C.

- A large collection of small languages that helped us investigate different aspects of ALOEs within simple environments.

Chapter 4

Action Routines

4.1. Introduction

Action routines are specified as part of the grammatical description of the language. One routine can be associated with each operator of the language. Action routines are used to perform context sensitive processing and to add language specific behavior to an ALOE. Action routines are called by ALOE in a variety of situations as described below. In this chapter, the application of action routines and their communication interface with the ALOE kernel will be discussed. The ALOE implementation environment, that provides the facilities and mechanisms for the implementation of action routines, will also be discussed. Action routines are optional, they need not be specified for all the operators of the language. A purely syntactic ALOE without action routines can also be generated.

The ALOE kernel communicates with the action routines by invoking them at the appropriate times. Action routines communicate with the ALOE kernel through their return values and through the invocation of functions from the ALOE implementation environment. Action routines communicate with other parts of the environment through explicit invocation of their functions and through modifications made to the common internal representation. The writer of action routines will be referred to as the ALOE Implementor.

4.2. Uses of Action Routines

Action routines are used for many purposes. The original design was intended to provide a mechanism for checking programming language semantics. However, in highly interactive environments such as the ones provided by ALOEs, action routines can be used to implement a large number of functions that some times have very little to do with programming language

semantics. Although the limits of their application have not been explored completely, they certainly promise a wide variety of possibilities, including:

- Programming language semantics checking. As said above, this was the first motivation for action routines, and is a very important application. ALOE can give the user immediate feedback on semantic errors as soon as they are made, giving him the opportunity to correct them immediately. The action routines can also provide some automatic correction themselves (*e.g.* automatic declaration of variables, spelling correction, *etc.*).

  Some semantic errors can be precluded by modifying the syntax of the language. In an ALOE environment, the user is explicitly specifying the language constructs that he wants to create, instead of having a parser infer them from examination of some text. So, the syntax can have *richer* constructs that would be difficult to disambiguate with a parser. For example, the use of "+" as a function name.

  In some languages, the syntax is disambiguated through semantics as is the case of the expression "f(n)" in ADA [DoD 80], which could be a function call or an array reference. In an $ALOE_{ADA}$ this is not an issue because the user will specify which type of construct he is building. The fact that the concrete representation is the same does not matter at all.

- Record state. Again, a lot of information is available because the environment is actively participating in the construction and modification of the programs. As the interaction with the ALOE takes place, status information can be recorded, updated, used, checked, *etc*. The ALOE implementation environment (see section 4.6) provides the ALOE implementor with a set of routines and data structures to help manipulate this state information.

  An important example of the use of state information is the implementation of some of the project management functions of GANDALF [Habermann 82], that synchronize the access of multiple users to the environment (see section 5.2.2).

- Interface to other parts of the environment. Every generated ALOE is an environment for a particular language. Action routines can be used to interface to the available tools of the environment. Some of these interfaces can be made transparent to the user and applied at the appropriate times. For example, a code generator can be invoked from the action routine associated with a particular operator (*e.g.* a procedure) when the user finishes entering the associated subtree. From the user point of view, compilation happens automatically. This is yet another example of the active participation of the environment in the programming process, and is not limited to construction and modification of programs.

  Another example is the interface with the display mechanisms, allowing the action routines to update the window allocation under certain circumstances (see section 4.3.3).

- Automatic generation of program pieces. There are a large number of cases in

which information already entered in the program can be taken advantage of and used to generate some pieces of program elsewhere. Some examples include automatic declaration of variables, derivation of specifications, initialization of default values, *etc.* In *aloegen*, the ALOE for creating grammatical descriptions (see section 3.2), entries for classes and operators can be made as soon as they are referenced.

This is one clear advantage of the interactive nature of ALOEs: the ALOE can anticipate what users may want to do and thus save time and effort. It is also a clear indication of the possibilities of exploitation of its language knowledge. However, every ALOE implementor may desire certain behavior for any particular language. For this reason, action routines are the proper mechanism to implement this functionality by giving the implementor the control over this behavior.

- Lexical analysis. ALOEs do a very limited form of automatic lexical analysis (see sections 2.4.1.2 and 3.2.2.1). As discussed in those sections, the automatic lexical analysis capabilities could be expanded. But action routines can also perform the lexical analysis when a terminal node is created and they can abort the creation if the token is illegal (see section 4.4). The lexical analysis performed by action routines need not be restricted to checking for the legality of tokens, but as semantic checking is coupled, the action routines can also check for duplicated names and other such kinds of name checking.

- General communication mechanism between different parts of the environment. In an integrated environment as the one an ALOE provides, the tools of the environment know about each other and can then cooperate towards a common goal. They can communicate with each other and transfer the necessary information through calls made, and data structures maintained, by action routines.

  One example of this communication is the one made between the code generator, the loader and the debugger in GANDALF [Habermann 79b, Feiler 82a].

If several of these applications will be handled in a single action routine, the separation and interaction among them must be handled explicitly in the action routine itself.

In an ALOE environment, action routines must deal with dynamic programs (*i.e.* programs that are constantly changing), as well as with incomplete ones (*i.e.* not yet fully specified). This contrasts with semantic checkers of compilers which deal with static programs: once the program is handed to them, it is complete and will not change. Action routines must then address and understand the impact on the semantics of a program when it is being constantly modified, action routines should also recognize when programs are incomplete and they should act accordingly.

Another important difference with semantic analyzers in compilers is the fact that programs may be semantically incorrect after an error is made and detected, and will not be necessarily corrected immediately. Deletion and modification of program pieces imply *undoing* of semantics (as well as other side effects, like the setting of defaults, *etc.*). It is then necessary to *broadcast* the effects of a modification. For a detailed discussion of how this is done in a particular programming environment see [Feller 82a].

In MENTOR [Donzeau-Gouge 80] semantic checking routines can be written in MENTOL and can be invoked by the user. This invocation must be made explicitly by the user instead of being done implicitly by the system as is the case in ALOE. The semantic checking that MENTOR provides is not automatic nor does it provide some of the other automatic processing that ALOE provides through its action routine interface (for example, the automatic generation of program pieces or the automatic invocation of other tools of the environment).

4.3. Calling Instances

ALOE activities are classified into different kinds of *actions*. There is a set of basic actions that include: CREATE for creation of nodes, INSERT and DELETE for insertion and deletion of subtrees and ENTRY and EXIT for moving in and out of nodes through cursor movement. When the **.nest** and the **.transform** editing commands were added as editing commands, the need for special actions was discovered and the NEST and TRANSFORM actions were introduced. When the windowing and context switching capabilities were implemented, the need for calls to deal with changing the contexts was discovered and the FAILUP and FAILDOWN actions were added. The calls on FAILUP and FAILDOWN are made when an unsuccessful attempt is made to move the cursor out of a node and into a node respectively. For every one of these actions there is a call on an action routine. Two parameters are explicitly passed on the call: the node where the action takes place and the *kind* of action. In this section each kind of action will be discussed.

The action routines are organized around these actions with the purpose of providing a general and flexible mechanism for the implementation of a variety of different environments. They could have been structured around the applications mentioned in the previous section but that would take away some of their flexibility by imposing a predetermined structure. It could also have an impact on their efficiency: lexical analysis or automatic generation of program pieces is not done everywhere. Finally, some of these applications were discovered

through the experience of using action routines and more applications will be found as more experience is gained.

No action calls are made in the *clipped* areas (see section 2.4.2.2). The clipped areas contain subtrees (pieces of programs) that are out of context and whose semantics could not be checked at that point. The semantics are checked, through an action call on INSERT, when a clipped tree is inserted.

4.3.1. Creation of Nodes

An action call on CREATE is made when a node is created as a result of a constructive command. A CREATE call on a non-terminal node can be used to automatically generate some fields (*e.g.* defaults), on a terminal node, it can be used to validate the value of the terminal node given by the user. When a non-terminal has just been constructed, its offspring are meta nodes. The CREATE action can be aborted by the action routine and ALOE will then *undo* the operator application and put the meta node back in its place. This is very useful particularly when the action routine is performing lexical analysis and determines that the token is illegal.

If the construction process of the tree would be *static* (*i.e.* once the tree is created, it is never modified), then the only action call needed is the one on CREATE. Indeed, this is the type of interface found in parser generators that include semantic routines [Johnson 75].

4.3.2. Visiting Nodes

An action call on EXIT is made when a node is left, going *up* towards its parent node in the tree, as a result of the cursor moving *out* of the node (see section 2.4.2.1). An action call on ENTRY is made when a node is entered from above, as a result of the cursor moving *in* to the node. Figure 4-1 shows a portion of a syntax tree indicating the direction of the cursor motion and the corresponding action routine call on ENTRY. Figure 4-2 does the same for the call on EXIT.

Communication with other pieces of the environment, such as access control, code generation, debugging, *etc.*, is achieved through the action calls on EXIT and ENTRY because they provide a convenient synchronization mechanism on the construction and modification process.

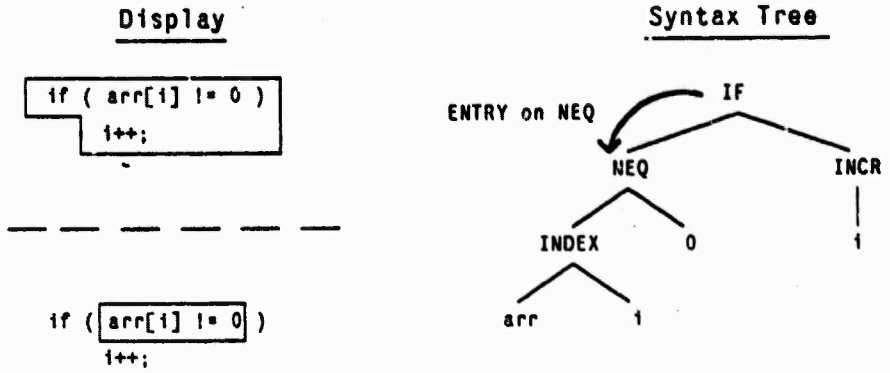

Figure 4-1: Action routine call on ENTRY

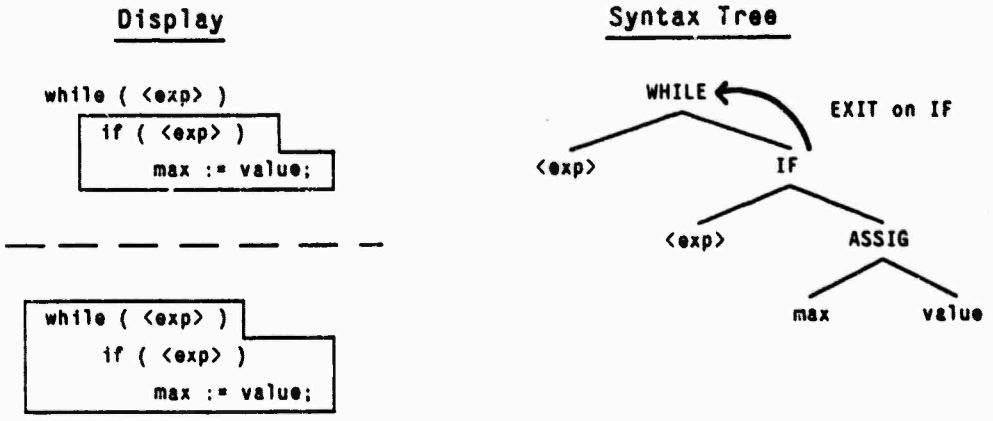

Figure 4-2: Action routine call on EXIT

4.3.3. Unsuccessful Cursor Movements

An action call on FAILUP is made when an attempt is made to move the cursor *out* of a node that is the *root* of the *current* window. This call gives the action routine the opportunity of updating the window allocation: it may cause a change of the current window, most likely a *pop* of the context window stack (see section 2.6). As part of the ALOE implementation environment, the window manipulation routines are provided.

An action call on FAILDOWN is made when the current node has no *visible* offspring as defined by the current unparsing scheme. This call gives the action routine the opportunity of

updating the window allocation and change the current unparsing scheme. It will most likely *push* a new window onto the context window stack with the current node as the *root* of the window and a new unparsing scheme so that the offspring are now visible. This manipulation can be viewed as a change of *context* in the program.

It has been observed, however, that the action routines follow a very uniform behavior in the FAILUP and FAILDOWN cases, namely the opening and closing of new contexts. This behavior could be abstracted and provided automatically by ALOE. The grammatical description would include information as to which operators will be *roots* of context windows together with the unparsing scheme to use when entering a new context. These changes would make the calls on FAILUP and FAILDOWN obsolete, making the writing of action routines much simpler. Keeping the calls as they are, gives the ALOE implementor more flexibility and control; but our observation is that it is not really needed for these cases.

The ALOE implementation environment provides a set of routines to implement the mechanisms for static and dynamic access control (see section 4.6.2 below), which are normally invoked from ENTRY and EXIT action calls. It has been suggested [Notkin 81a] that the access control mechanism could also be abstracted and incorporated into the grammatical description. This again makes the writing of these action routines much simpler at the expense of some flexibility and control, on the other hand, it increases the complexity of the grammatical description. But if the specification language is rich enough to express the desired behavior, action routines for these mechanisms are no longer needed.

As more experience is gained with ALOEs and action routines, it is expected that more of these mechanisms and applications can be automated and expressed with appropriate specification languages in the language description.

4.3.4. Tree Transformation Operations

An action call on DELETE is made just before a subtree is deleted. This is to allow the action routine to use all the information contained in the subtree before it is lost. The return value from a DELETE call can indicate that the deletion should be aborted, in which case ALOE will not perform the deletion.

In the case of DELETE, as well as with the other calls caused by tree transformation

operations, semantic information (as well as other information and side.effects caused by the construction process through CREATE calls) must be *undone*. The effects of the modification must be reflected in related nodes so that the proper and consistent state can be kept. These effects must be propagated explicitly by the action routine. For different types of nodes and different kinds of operations, these changes will require varying amounts of processing. With respect to semantics, it is not the same to delete a constant that to insert a large subtree.

An action call on INSERT is made when a node from the clipped area is inserted. Although it should be similar to the CREATE call, in this case the action routine may not want to set default values or do other automatic generation. The insertion process works top down in the subtree with an INSERT call being made at every node as it is inserted. So, for non-terminal nodes, at the point of the call, its offspring are meta nodes as in the case of the CREATE call, but those offspring will be immediately filled in after the call returns, as the insertion process continues.

An action call on NEST is made when a subtree is being nested into a new subtree which will replace the original one. The **.nest** command (see section 2.4.2.2) is only allowed by ALOE when the nesting and the resulting subtree are syntactically correct. Depending on the language and the application the NEST call may behave as an implicit CREATE, since after all, there is a new node being created.

In terms of the syntactic structure of the tree, the effect of the **.nest** command can be achieved with the following sequence of basic tree modification commands: a **.clip** and **.delete** of the current subtree, followed by the construction of the new node, followed by the **.insert** of the clipped subtree. Even though users learn very rapidly to think in terms of these basic operations, as evidenced by the experience with the use of different ALOEs and with the Synthesizer, it is not an efficient way to achieve the desired effect. From the point of view of action routines it is even worse because the DELETE call will then have to assume that the subtree is really being thrown away with all its information and the effects must be propagated and the corresponding nodes invalidated, just to have the set of INSERT calls reconstruct all these information again. With the **.nest** command, all information contained in the subtree is kept and in most cases will probably not change. Figure 2-4 shows the effect of an application of a nest command.

An action call on TRANSFORM is made after a transformation is performed on a subtree by

mapping it into another, by changing the root node. The **.transform** command (see section 2.4.2.2) is only allowed when the offspring of the original node and the new node match. The TRANSFORM call may perform the functions of a CREATE call on the new root. The difference with the creation is that the offspring of the node are not meta nodes, so only a subset of the CREATE functionality would probably be desired. Figure 2-5 shows an example of the application of a **.transform** command.

As in the case of **.nest**, the effect of the **.transform** command can be achieved with the following sequence of basic tree modifications: first, the offspring of the current node are clipped and deleted; then, the current node is deleted; then, the new node is constructed and the clipped subtrees are inserted back again. Again, it is very cumbersome for the user to achieve the desired result. The action routines would not be aware of the kind of transformation that is taking place and would have to do more than the necessary processing. The calls on NEST and TRANSFORM are more efficient because they utilize the information that is available and that would otherwise be destroyed by the DELETE calls and reconstructed by the CREATE and INSERT calls if the basic commands would have been used.

An action call on TDELETE is made on the original node before the transformation takes place but after it has been determined that it is a legal one. The call is necessary because the root node will be deleted and substituted by a new node. The TDELETE call can also be aborted in which case ALOE will not perform the transformation. Information contained in the node may be used and any side effects must be taken care of. It is different than the DELETE call because the subtree will not be deleted and information contained there may be kept or updated accordingly.

An action call on EDIT is made after a constant is changed by an **.edit** command which invokes a text editor for editing text constants (see section 3.2.2.1) because ALOE lacks support for underlying text editing. This call allows the action routine to validate the value of the constant (as it would do in the case of a CREATE). As the only changing structure is the value of the constant, the node itself does not change at all. One possible action that may be desired, is that of recording that a change was made in a particular subtree.

If new tree transformation operations are added to the set of editing commands of ALOE, corresponding action calls must also be added, so that the action routines can take full

advantage of their knowledge about the kind of operation that is taking place and manipulate the information and state accordingly. If these transformations are added through extended commands, then they apply to a particular ALOE only and the implementation of those extended commands will take care of the semantic effects. No general mechanism needs to be provided in that case.

Action routines are used to maintain certain state information in the nodes of the tree. This state information can be used for many purposes including access control, to indicate if a subtree is correct, if there is an error or if it should be checked, *etc*. It is important that ALOE makes a consistent sequence of calls every time a node is visited: for every ENTRY call, there should be a corresponding EXIT call. This would guarantee that if the ALOE implementor wants to keep the information in a stack of some sort, the number of *push* operations (on ENTRY) will match the number of *pop* operations (on EXIT), as the cursor is moved around the tree.

For the purposes of this consistency, a call on CREATE or INSERT should also perform the necessary functions of an ENTRY call, since no separate ENTRY call is made. Since all the information is available to the action routine , the extra call is not necessary and this also helps the efficiency of the action routines interface. Similarly, calls on DELETE or TDELETE should behave as an implicit EXIT. The call on NEST should behave as an EXIT on the original subtree and as an ENTRY on the new node (maybe through a CREATE). The call on TRANSFORM should perform the necessary functions of an ENTRY on the new node (again, maybe through a create).

4.4. Return Values from Action Routines

The communication between action routines and the ALOE kernel is achieved through the return values of the action routines, as well as through calls to routines provided as part of the ALOE implementation environment described in section 4.6 below. The ALOE implementation environment includes an error reporting interface that provides the means of communication between action routines an the user. There are several possible meanings that return values from action routines can have:

- Abort the action. Applies to the CREATE, DELETE and TDELETE calls. In the CREATE call it means that the operation should be *undone* and a meta node should be placed back in that position. The program cursor is positioned at the

meta node. In the DELETE and TDELETE cases it means that the operation should not take place. The program cursor is not moved.

- Continue. The program cursor will be positioned where it would have been positioned if there were no errors. If errors occurred, they are shown to the user but the program cursor is not repositioned.

- Redirect the program cursor to the node of the last error reported. In general this is used to let the user select the error he wants to correct next. In the absence of errors this and the previous case are identical.

- Redirect the program cursor to a specific node. This is used to force the user to a particular node as a result of some action. An example of its use is for access control. In a case in which a particular node in the tree should not be entered by some user, the ENTRY action call can redirect the cursor to the parent node, thus effectively preventing the entry to the node. An error message can be generated to explain the reason for this action.

- To indicate that the current node (the node where the call is made), has been replaced by the action routine.

In some of these cases, the program cursor will be redirected to a different node than the one where the call was made. In order to keep consistency, ALOE will find the path from the original node to the resulting one and will make all the necessary EXIT and ENTRY calls. If the original node was replaced, the path is found starting at the replacement node.

4.5. Error Reporting

If there are errors generated by the invocation of an action routine, ALOE will display them to the user one by one. After every error is displayed, the user has the option of stopping at that error or looking at the next error. This error reporting mechanism provides the communication from action routines to the user. Its flexibility allows the user to stop at any particular error to correct it.

When a user decides to stop at an error, the remaining errors are not kept by ALOE. ALOE does not understand the content or the kind of error (it only knows the difference between errors, warnings and simple messages). ALOE would not know if an error that is pending is corrected by a modification that the user makes. ALOE assumes that pending errors will be regenerated if the modifications do not correct them.

One problem with this scheme is that situations may arise in which the same error message

will be produced over and over again until the user corrects it. In some instances this may be undesirable and the action routines may want to *filter* some errors already displayed and not display them until the user leaves some context. For a discussion of one such error filtering mechanism, see [Feiler 82a].

4.6. The ALOE Implementation Environment

The ALOE implementation environment provides an environment for the implementor of action routines and extended commands. It provides a data encapsulation mechanism for the internal representation that defines the data structures that are accessible as well as the operations that can be performed on them. These operations provide the facilities for inspection, traversal and modification of the internal tree representation. They guarantee the syntactic correctness and integrity of the internal representation. The ALOE implementation environment actually provides an operational definition of the internal structure. Using this environment, the task of implementing action routines and extended commands is much simplified and *secure* (modifying the internal tree representation in undesired ways, is precluded). The C programming language [Kernighan 78] does not provide enough protection mechanisms to enforce this mechanism. So, the integrity of the internal representation can be guaranteed only if it is accessed through the ALOE implementation environment.

The ALOE implementation environment also provides mechanisms for other operations such as access control, error reporting, tree traversal, window manipulation, status checking, *etc*. The following sections discuss the main characteristics of these operations. For a detailed description of the specification of the primitives provided by the ALOE implementation environment, see C.

4.6.1. Error Reporting Interface

The ALOE implementation environment provides the primitives for the general error reporting interface for action routines described in section 4.5 The error interface provides the communication between the action routines and the user. The interface recognizes three kinds of messages: errors, warnings and plain messages. The display and window facilities of ALOE (see section 2.6) are coupled with the error interface to provide a better user interface. The error interface buffers all messages caused by an action routine call and then displays them, indicating the kind of message and highlighting the associated node.

4.6.2. Access Control

The implementation environment provides a set of access control primitives for restricting and enabling editing commands at different levels of the development of the internal tree. This includes environment specific extended commands. Constructive commands are restricted or enabled as a group, no control of individual commands is provided. It would be very desirable to be able to have access control over individual language commands as well. A very interesting application of this form of access control would be in teaching a programming language. A small subset of the most important language constructs of the language is made available in an ALOE for the full language. As the user learns the new language, more language commands are made available. Until finally, the whole language is made available. The help facility in ALOE is coupled with access control, so only the available commands are displayed when help is requested. In the learning process, the user interacts with an ALOE with the same user interface.

The primitives for access control communicate with the ALOE kernel by setting an attribute associated with every editing command that determines whether or not the command is legal. The kernel checks the value of this attribute before executing the command. The ALOE implementor does not access this attribute directly but does so through the set of action control primitives.

ALOE maintains a stack of access control words, each of which controls the set of legal commands. When a certain level of the tree is entered, an action routine can *push* a new access control word into the stack, thus redefining the set of legal commands for the new level. When the subtree is left, the action routine can then *pop* the stack to reset the access control state to where it was before entering the level.

As the user moves in and out of subtrees, the action routines use the access control primitives to permit or restrict some commands, thus *shaping* the accepted behavior of different users of the environment. Different users may have different kinds of *rights*.

These access control primitives provide the action routines with the mechanisms to implement static and dynamic access control. Static access control can be set by defining the set of commands that are legal upon entering the system root. Dynamic access control is implemented through the calls to the primitives that are manipulating the set of legal commands.

Access control is primarily used for project management [Habermann 82] that synchronizes the access to a system by different users with different access rights. For further discussion on project management and access rights, see section 5.2.2.

4.6.3. Tree Traversal

The ALOE implementation environment provides a set of routines to *traverse* the internal trees. Access to parents nodes and offspring (*i.e.* internal tree pointers), is not done directly but through a set of tree traversal primitives that allow the action routines to move through the tree and get to desired nodes.

These primitives make file nodes (see section 3.2.8) transparent to the ALOE implementor. ALOE provides automatic checkpointing of the files associated with subtrees through file nodes. This checkpointing is done whenever a file node is passed through, either by going from a node to its parent node or to one of its offspring nodes. This passing is referred to as a context change. When a context is changed, if the subtree has been modified, it is written out.

These mechanisms, coupled with the ability to partition the data base (or program) into small separately stored files through file nodes, provide a very convenient checkpointing capability but suffer with respect to recovery, that is, the ability to *undo* commands [Archer 81a], because files are automatically overwritten quite frequently (depending on how much of a partition is specified).

Other tree traversal primitives are provided to action routines for *browsing* through the tree with no context kept and no checkpointing done. They provide a more efficient mechanism to move around the tree when no modifications will be made.

There are some instances in which the action routines need access to the file nodes themselves. Sometimes it is desired to stop at a file node without reading in the subtree associated with it. Also, symbol table implementation needs access to file nodes because symbol table files can also be associated with the file nodes. For these purposes there are specific primitives to give access to file nodes. For further discussion on the advantages and disadvantages of file nodes, see section 5.2.6.

4.6.4. Window Manipulation

Ad discussed previously in section 2.6, there is not enough screen space to display all the necessary information in an ALOE environment. The screen organization is then critical to the efficient utilization of the available resources. The ALOE kernel uses a sophisticated display and window interface to manage the screen [Feiler 81]. Some of the window manipulation facilities are made available to the ALOE Implementor through the ALOE implementation environment. Among these facilities is the stack of context windows maintained by ALOE. The ALOE implementation environment provides a set of primitives for manipulating this stack. Using these primitives, the ALOE implementor can then divide his structure (or program data base) into different contexts and have them displayed in overlaying windows.

If the ALOE implementor is developing a programming environment in which a user is going to run programs, he needs a separate section in the screen for the input and output of user programs. As part of the display interface of the ALOE kernel, a *user* window is provided for these purposes. The implementation environment provides the necessary primitives to manipulate the user window.

4.6.5. Status Manipulation

Every node in the internal representation contains an extra field for status information. A typical use of the field is to put semantic information in it. This status information is stored in the files written by ALOE, thus saving semantic information that can be used again and does not need to be recomputed. The implementation environment provides a set of primitives to implement a simple scheme for status manipulation that helps in setting and resetting status values as well as broadcasting changes. If the action routines implementation requires a more complicated status manipulation mechanism [Feiler 82a], the status fields in the nodes are accessible through the ALOE implementation environment.

4.7. Extended Commands

The ALOE implementation environment is also available for the implementation of extended commands. The implementation of these commands can also invoke action routines and use the same action routine communication interface the the ALOE kernel uses. Figure 4-3 shows the implementor's view of the ALOE system.

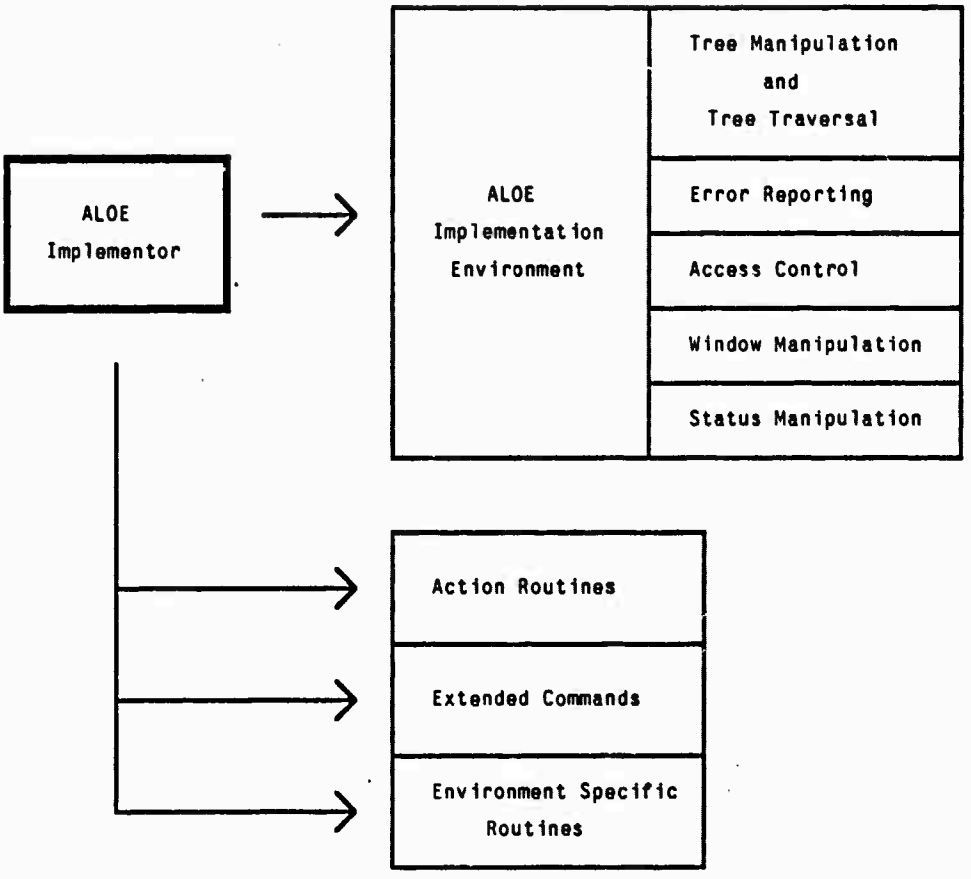

Figure 4-3: An implementor's view of the ALOE system

4.8. Attribute Grammars

As discussed earlier in section 4.2, the motivating application of action routines is checking programming language semantics. It would be very desirable to express those semantics using a formalism like attribute grammars [Knuth 68].

Attribute grammars have been traditionally used in compiler generators [Nestor 81, Ganzinger 77, Rahia 77]. But compilers deal only with static programs: once the program is entered, it never changes. In an ALOE environment it is necessary to deal with dynamic programs that are constantly changing as well as with incomplete programs. This situation arises due to the interactive nature of an ALOE environment in which the user interacts directly with the environment in the process of constructing and modifying his program.

The important implementation issue is that of *undoing* semantics and the propagation of changes. When the user makes a change to his program, some semantic knowledge or structure already built must be modified or removed. The effects of a modification are rarely localized in a particular node. In most instances the effects must be propagated to other nodes so that other information and status that is related to the modification can be updated.

In an ALOE environment it is also necessary to deal with semantically incorrect programs for various periods of time: for instance, the user has been notified of his errors, and either he decides to correct them later or he is in the process of correcting several of them. In a compiler environment, when a semantic error is found, the program is illegal, or if the error is a simple one, and the compiler is smart enough [Graham 79], the compiler will correct it and the program is no longer semantically incorrect.

The use of attribute grammars for syntax-directed editors has been proposed in [Demers 81]. An algorithm is proposed for the reevaluation of attributes given a subtree replacement. An optimal algorithm is proposed in [Reps 82]. One of the most important aspects of an evaluator in an interactive system is that it must be efficient (*i.e.* it cannot cause any significant delay in the response time of the system). If a change is made that affects an attribute of a node that is not in the vicinity of the changed node, the corresponding attributes of all the nodes in the path between the two nodes must be reevaluated. This is because attributes can only depend on attributes from the parent node or from one of its offspring. Reevaluation of attributes is not the only task to be performed, but also the determination of the set of attributes that must be reevaluated, but this is handled implicitly through the attribute relationships instead of having to be done explicitly as with action routines.

An implementation of the algorithms has been done [Reps 81]. The use of these algorithms in a generator of environments will probably answer these efficiency concerns, especially in dealing with symbol tables whose handling in attribute grammar systems has been traditionally very inefficient.

To address some of these efficiency concerns, Johnson and Fischer [Johnson 82] propose the modification of attribute grammar formalisms to include non-local attributes and attribute flow relations that are not confined to the parent or the offspring of a node. Using this model, direct attribute links can be provided between definitions and uses of variables. These links can be provided in an ALOE environment through the symbol table manipulation interface and action routines [Kaiser 81].

Related also to the efficiency issue is the impact of the tree transformation commands such as **.nest** and **.transform** (see section 2.4.2.2), on the semantic checking. It is true that these commands can be replaced with a series of **.clip**, **.delete** and **.insert** commands. However, most of the semantic information already in the tree can be kept. For example, in the case of **.nest**, the subtree is really not deleted at all (an extra node is inserted in the tree), and thus, the semantic information need not be *undone*. The same is true for **.transform** and any other tree transformation operation that could be added. In section 4.3 we discussed the need for a NEST and TRANSFORM action routine calls to take advantage of the information provided by the type of operation that is taking place. The attribute grammar mechanisms would have to be modified to be able to take the same kind of advantage. This implies that subtree replacement, as proposed in [Reps 82], is not necessarily the correct unit for attribute reevaluation.

With attribute grammars it is still necessary to provide a semantic function for every attribute. In many cases the code in an action routine is basically only the equivalent of attribute evaluation. Writing action routines is made much easier with the ALOE implementation environment. It is not clear that writing the attribute grammar is easier than writing the action routines.

In section 4.2 we discussed the many applications of action routines. As discussed above, attribute grammars (or a suitable variant) could be used to help in automating the generation of more sophisticated integrated programming environments. Attribute grammars, as they have been traditionally considered, may not be a powerful enough formalism for some of the applications discussed. One such application is the automatic generation of program pieces and in general the ability to modify the internal structures. Action routines have this ability through the ALOE implementation environment. Attribute grammars have been traditionally considered as a formalism for language recognizers and not for language generators. Another application is the ability of an action routine to abort a deletion of a subtree to provide one form of access control. This functionality is not available in an attribute grammar system.

Attribute grammars implicitly specify the dependencies between operators and evaluation order information, while action routines have to explicitly implement them. Action routines provide a very flexible mechanism but with little control. Some synthesis of both approaches is worth investigating as a natural extension of this work.

Chapter 5

Building a Large Integrated Environment: The GANDALF Environment

5.1. Introduction

The ALOE implementation environment, described in section 4.6 lets the implementor add the environment and language dependent functionality needed to develop large integrated environments. The key feature of an ALOE that permits this development is the uniform user interface that, together with the common program representation, allows the integration of the environment pieces. The GANDALF software development environment [Habermann 79b, Habermann 82, Notkin 82b] is the most complex instantiation of an ALOE environment built to date. In this chapter we discuss the different kinds of support provided through the ALOE Generator and the ALOE implementation environment for the development of such large integrated environments, focusing the discussion on the development of the GANDALF environment.

The characteristics that make GANDALF a software development environment rather than just a programming environment are the incorporation of system version control [Kaiser 82] (referred to elsewhere [DeRemer 76], as *programming-in-the-large*) and project management that synchronizes the access to the system data base by a group of members of a project with different access rights [Habermann 79a] (referred to elsewhere [Notkin 81b], as *programming-in-the-many*).

All this functionality is incorporated in GANDALF through the mechanisms of an ALOE. System version control is incorporated through a language description of the version control structure of the system and through some extended commands. This structure is checked

and validated through action routines. Project management is incorporated through the addition of some extended commands and through action routines that implement some of the project management functions automatically and transparently to the user. Documentation is also added to a GANDALF database structurally as operators of the language. Documentation operators are terminal operators of type *text* constant (see section 3.2.2.1) which are entered and edited using a text editor that is invoked by ALOE for that purpose.

5.2. ALOE Support for Large Integrated Environments

The generated environments have four important properties:

- Uniform User Interface. All interactions with the environment are done through the user interface of ALOE. Users thereby have a single language for communication with the environment instead of a collection of different languages or interfaces to communicate with the different pieces or tools of the environment. The user need no longer be aware of the specific invocation of a particular tool but instead he concentrates on the tasks he needs performed (*e.g.* program editing, program execution, system description, *etc.*).

- Integrated. All pieces of the environment are now knowledgeable of each other and can collaborate towards a common goal, instead of being a collection of independent tools [Dolotta 76, Ivie 77]. This integration is supported in an ALOE through the use of a common program representation that the different pieces access and modify. Invocation of certain functions of the environment is now implicit and transparent to the user. For example, the compiler need no longer be explicitly invoked by the user but rather the environment can invoke the code generator every time a compilation unit (*e.g.* a procedure) is completed by the user.

- Incremental. As the user progresses through the development of his programs the environment can collect information that allows it to perform incremental checking and updating. For example, after every construct is entered it can be semantically checked, code generation can be performed after a procedure is completed, common defaults can be automatically generated as constructs are entered, *etc.*

- Interactive. Every ALOE generated is an interactive system. The incremental nature of the generated environments makes it possible to process small pieces of program at a time instead of having to process the entire program. ALOE takes advantage of the users' *think* time at the terminal to perform this processing.

The ALOE facilities that provide the functionality to support the development of large integrated environments are: extended commands, access control, action routines,

environment specific routines, and multiple concrete representations. This support gives the ALOE implementor the ability to incrementally develop an environment. He can start with a simple syntax-directed editor and then incrementally expand it through the implementation of action routines and environment specific routines and adding extended commands. In this manner the complexity of the environment can be increased in a controlled form.

The rest of this section discusses separately each one of these different mechanisms with respect to the support they provide for the development of large integrated environments.

5.2.1. Extended Commands

The basic set of editing commands discussed in section 2.4.2 can be extended with commands that invoke environment specific functions through the same uniform interface of ALOE. Typical uses of extended commands are for communication with other pieces of the environment. Commands in GANDALF such as **.run** and **.continue** communicate with the run-time environment.

Project management functions [Habermann 79a, Habermann 82] which provide, among other mechanisms, synchronization between multiple users of the environment, are invoked through commands such as **.reserve** which makes a particular unit modifiable only by the programmer making the reservation and **.deposit** which makes the unit available again for other programmers. Commands such as .revise which creates a new revision of a version is an example of a system version control function [Kaiser 82] implemented via extended commands.

In a simple stand-alone ALOE, searches done with the **.find** command (see section 2.4.2.1) search through the whole program tree. In more complex environments **.find** only applies to the current *window* or *context*. For these environments it is useful to include environment specific searching commands that understand the structure of the program data base and can focus the search using this knowledge. For example, in GANOALF a search for a procedure looks within the current version and revision instead of the first occurrence of the procedure in the program data base. GANDALF search commands understand the structure of GANOALF data bases.

The complexity of the implementation of extended commands depends directly on the

complexity of the function to be performed. The ALOE implementation environment provides the basic facilities for accessing and modifying the data base, but cannot predict the kinds of operations the implementor might want. This allows the ALOE implementation environment to provide flexible facilities for the development of a wide variety of functions.

Commands such as environment specific searches are straight forward to implement and amount to a *knowledge directed* simple tree searching. On the other hand, commands like **.deposit** are not so trivial because they have to check and validate large parts of the data base before allowing the actu deposit to take place.

5.2.2. Access Control

The access control facilities allow the ALOE implementor to provide different levels of user *rights* for access and manipulation of the program data base. For example, in GANDALF, only the *project leader* can modify the list of programmers that have access to the data base. This is an example of *static* access control. The status of the program data base can also be used to decide whether or not to permit certain operations. For example, a data base that has already been *deposited* (i.e. made public) cannot be edited again. Another interesting example is in a list of log messages, where the commands **.extend** and **.append** are restricted and only **.prepend** is permitted. This guarantees a chronological order (most recent first) in the logs. These are examples of *dynamic* access control.

As discussed in section 4.6.2, the implementation of access control is done through a vector of access rights with one entry per editing command. The implementation of access control policy for a particular environment is rather straight forward through the use of the basic set of routines that manipulate the access control vectors provided by the ALOE implementation environment. These routines are normally called upon initialization to set up the *static* access control based on the rights of the user and from action routines as certain nodes are *entered* or *exited* to set the *dynamic* access control.

5.2.3. Action Routines

In section 4.2, the many possible applications of action routines were discussed. Some of these applications were anticipated and were the motivation for the design of action routines. These include semantic checking and implicit invocation of other tools in the environment such as the code generator. Other applications were discovered in the process of developing different ALOEs especially GANDALF. Examples of these include automatic generation of program pieces, lexical analysis, interaction with display management, *etc*.

The ALOE implementation environment, discussed in section 4.6, provides a flexible but controlled and safe access to the internal representation as well as mechanisms to perform common operations such as tree traversal, error reporting and display management, among others.

The design of the action routine interface, discussed in section 4.3, gives the ALOE implementor full control in the process of building a program or data base. Action routines are called whenever nodes are created, deleted, transformed or visited. In addition, there are two special kinds of action routine calls: when an attempt is made to leave a node that is currently the *root* of the current window (a FAILUP call) or an attempt is made to enter a subtree that is not currently *visible* (a FAILDOWN call). In the first case, the action call allows the implementor to reassign the root to an ancestor node and to change the window to a previous context. In the second case, it lets the implementor change the unparsing scheme a. allocate a new window for a new context. The ALOE implementation environment provides the routines for these window manipulations.

The action routine interface also defines certain mechanisms for communication from the action routines to the ALOE kernel (see section 4.4). One of them indicates, for the cases of creation and deletion, that the operation should be aborted, that is, that the creation should not be allowed and the meta node should be put back in its place or that the deletion should not take place.

The action routine interface permits and supports incremental development of programs or data bases by being able to process small units at a time, due to the fact that action routines are invoked almost after every interaction of the ALOE. Every action routine invocation only needs to do a small part of the task allowing the environment to be really interactive by having the delay per interaction not too large.

As with extended commands, the complexity of the implementation of action routines depends directly on the complexity of the desired functionality. The ALOE implementation environment cannot predict all the operations that the ALOE implementor wants to perform. Simple operations such as status checking, that only operate on the node the action routine is called on, are very easy to implement. Other operations that must traverse and update values at many nodes within the data base are more difficult to implement. Some other operations depend on the existence of support from the other pieces of the environment that are being integrated. For example, if an incremental code generator is not available, then the complexity of providing incremental code generation is not in the action routines interface but in the implementation of the incremental code generator itself.

Some specific examples in GANDALF of the notions and applications discussed in this section include: automatic generation of program pieces is done, among other instances, when a log message is created, the user id and the date of the log are automatically entered in the log subtree. On exiting from the list of leaders of a project, which is part of the GANDALF data base, if the list is empty an entry is automatically generated by the action routine, thus avoiding leaderless projects.

The ability to abort a construction on a CREATE call is used to avoid duplicate entries in several lists, such as the leaders and programmers lists. The same ability in the DELETE call can be used in the case of the leaders list to prevent it from becoming empty.

Typical examples of operations done normally on *exits* from nodes include: code generation, consistency checks, removing of unnecessary meta nodes, stopping lists from becoming empty, *etc*. Most of these functions are then transparent to the user, unless an error occurs. This means that the user does not have to worry about this function explicitly any more. The support given in ALOE for these functions is very flexible, and indeed these functions do not have necessarily to be invoked all the time. For example, code generation at procedure exit may not be invoked until the procedure is fully specified in one case, or it is invoked always if the ALOE implementor wants to provide support for incomplete programs. In either case it is entirely up to the ALOE implementor to make this decision.

5.2.4. Environment Specific Routines

Certain mechanisms of an ALOE require different functionality for different environments. Some examples of these mechanisms include the symbol table manipulation, mapping of data bases to the file system, window layout definitions, initialization and clean up operations, *etc.*

ALOE provides a set of redefinable routines, invoked by the kernel, that implement these mechanisms. For a complete description of the set of redefinable environment specific routines see [Medina-Mora 81a]. ALOE provides a set of default implementations for these routines that provide simple versions of these mechanisms. The ALOE implementor can redefine some or all of them to provide the desired environment specific behavior. For the implementation of these routines, the ALOE implementor can use the ALOE implementation environment. In many cases the ALOE implementor is able to simply modify the default version of a routine, which makes his task easier.

For a simple stand-alone ALOE the symbol table manipulation need not be anything more than a simple name table scheme for unparsing. For environments that need to perform semantic checking or code generation through action routines, a more sophisticated symbol table manipulation is necessary. The design of the GANDALF environment includes a multi-level symbol table, which consists of three levels of symbol tables. At the top level there is a single symbol table called the *global* symbol table which contains an entry for every BOX or MODULE. There is one *module* symbol table for every MODULE and finally at the lowest level there is one *local* symbol table for every compilation context. For a detailed description of the GANDALF symbol table, see [Kaiser 81].

The symbol table manipulation mechanisms were the motivating force behind the design decision of allowing redefinable routines for certain mechanisms. It became evident, when more sophisticated environments were developed, that the simple symbol table manipulation mechanisms provided for a stand-alone ALOE was not enough for other environments, especially in the case of GANDALF where a multi level design was desirable. But it was also clear, that all the complexity of the GANDALF symbol tables was not necessary for simple ALOEs. It is likely that in the process of developing other sophisticated environments, it will be found that the redefinition of some other mechanisms should be allowed, to provide the desired functionality.

5.2.5. Display Management

Any ALOE user has some control over the layout of the different predefined windows on the screen, described in section 2.6. When more sophisticated environments, such as GANDALF are developed, it becomes necessary to provide more types of windows than in a simple ALOE. Examples of such windows include windows for debugging information, such as a window for monitoring values of variables and a window to display the call stack structure of a running program. There is also a need for a *user* window for input/output of the user program being developed using GANDALF. A full description of the ALOE window manipulation interface can be found in [Feiler 81].

With this expanded set of windows the screen organization becomes much more important and the physical limitations of the screen have a greater impact. The context window stack (see section 2.6) also becomes very important and is heavily used in environments like GANDALF. The ALOE implementation environment provides the support for manipulation of this stack from action routines.

Two special action routine calls were added to the existing set of calls for this window management. They are the FAILUP and FAILDOWN calls, described in section 4.3.3, which are used by the ALOE implementor to allocate and deallocate windows in the context stack, so that the current context is always the one that is displayed. The context stack window is very helpful in showing the current nesting of contexts but takes away one line from the program window (where the context windows are overlaid). Similar trade-offs exist for other kinds of windows: useful information *vs* space restrictions.

One big design problem is that of the user window. As discussed above, it is used to provide input/output for the user program being developed but it is a window and not a full screen as any user program would normally assume. Difficulties arise if the user program wants to make any use of the screen capabilities directly. On the other hand, to provide language oriented debugging [Feiler 82a] it is necessary to share the screen between program development and program input/output. A solution to this problem was proposed in the Copilot system [Swinehart 74] through the use of multiple screens. Better display technologies will constitute a big step ahead by being able to provide full screen functionality through independent *virtual* windows or displays in one screen as is done in AT [Ball 80], Canvas [Ball 81] and in the display oriented Interlisp [Teitelman 77, Sproull 79].

5.2.6. File Nodes

In stand-alone ALOEs and for simple environments, the developed programs are stored in a single file. In more sophisticated environments such as GANDALF, the user is actually building large data bases that include many programs. The need for a partition of these data bases into separate files, became evident. The concept of *file nodes* was then incorporated into the design of ALOE (see section 3.2.8).

The ALOE implementor determines the partitioning explicitly in the grammatical description by indicating which operators should be roots of separately stored files. In the internal representation, a separate file node is inserted on top of these special nodes upon their creation. The file node contains information about the name of the file where the associated tree is stored. When reading a subtree, the file nodes appear as leaves of the subtree with the actual subtree stored in a separate file.

This partitioning of the data bases in separate files makes checkpointing of files rather easy and is automatically handled by ALOE. Every time a user leaves one of these subtrees, if the subtree has been modified, it is written out. Another advantage is that the whole data base is never read when a session begins, but its different pieces are read as required. This has the advantage that although previous versions of a module in a GANDALF data base are part of the data base and of the logical structure, they are hardly ever read unless a user specifically wants to look at them. This keeps the in-memory size of data bases well below the actual size of the data base with the obvious advantages in efficiency. The traditional criticism of syntax-directed editors of using too much space [Morris 81], is no longer necessarily true.

File nodes can point to symbol table files that will also be stored separately. This feature is extremely convenient for multi level symbol tables such as the GANDALF symbol table. Some of the operators where the partitioning takes place coincide with the logical levels of the symbol tables. For example, in GANDALF the file nodes associated with a *module* point to the symbol table file for the module symbol table.

It is possible to couple file nodes with visibility rules and contexts so that every operator that is a root of a separately stored subtree can also be the root of a subtree in a window. As the user moves into one of these subtrees, a new window can be allocated to display the subtree, defining a new context. However, contexts are not bound to the partition determined by the file nodes. In GANDALF there are contexts that get opened at points in the structure

that are not file nodes. On the other hand file nodes and contexts work together in a natural manner.

The original design goal of file nodes was to make them totally transparent to the user, and in most cases, they are. It was discovered, during their implementation, that file nodes were legal cursor positions for the cases in which they appear as leaves of the subtree of a previous context, when the associated subtree has not been read yet. A **cursor-in** command opens the new context and moves the cursor to the associated node. If no new context is associated with the file node, then the file node is not a legal cursor position and is then totally transparent to the user.

In figure 5-1, we can see that in the BOXES context, the file node associated with module BasicOps is the current cursor position. After a **cursor-in** command, the MODULE BasicOps context is entered and the cursor is at the module subtree.

The ALOE implementation environment provides a set of routines for tree traversal (see section 4.6.3) that *hide* file nodes from the ALOE implementor and perform automatically the reading and writing of the associated subtrees. For the cases in which there is a need to get to the file node directly, especially for those action routines that implement functions related to the symbol table manipulation, the ALOE implementation environment provides the necessary routines to access the file nodes.

In the original design of file nodes, it was also intended to *hide* them from the ALOE implementor except for the cases in which he explicitly wanted access to them as described above. The idea was to handle them only through the set of tree traversal routines mentioned above. Their internal structure design is different from the rest of the tree nodes (for details of their internal structure see [Medina-Mora 81a]). This turned out to be a big flaw in their design. During their implementation it was decided that file nodes could be legal cursor positions for the cases described above, and thus they no longer were hidden from the rest of the ALOE kernel and the ALOE implementation environment.

All the other tree nodes have a similar internal structure and can be handled in the implementation code in a uniform manner, and only those routines that need to differentiate among them will access the fields of the nodes according to their specific structure. But file nodes with their different structure must be handled separately and in many parts of the implementation code special cases for dealing with them had to be introduced. Their

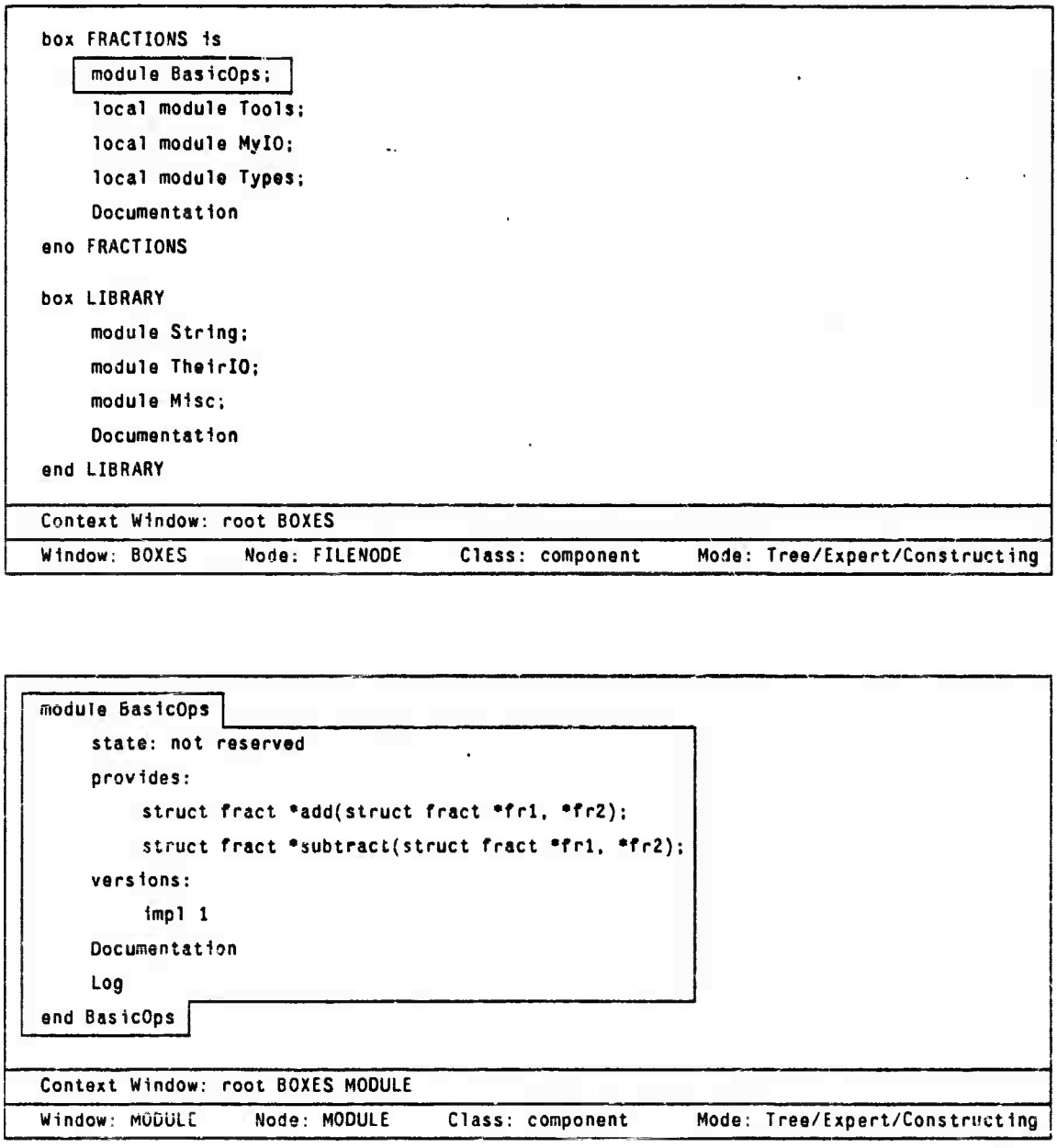

Figure 5-1: Effect of a cursor-in command on a file node

implementation had an impact on the implementation of almost every aspect of ALOE, thus making their incorporation difficult. If the operating system provides a good virtual memory implementation, the need for file nodes disappears, because the operating system will read the pieces of the data base only when they are explicitly referenced.

5.2.7. Parser Interface

An ALOE implementor may want to provide a parser for programs built with text editors so that they can be incorporated into the ALOE environment. Every ALOE, upon invocation can invoke a parser that would build an abstract syntax tree from the text representation of a program. These parsers must be written explicitly for any desired environment, they are not automatically generated. The GANDALF environment does not include a parser, there are no GANDALF data bases built as text files.

The automatic generation of parsers from the grammatical description is a natural extension of this research. It would be desirable to get a parser for the language for which an ALOE is being generated so that all previously built programs can immediately be incorporated.

5.2.8. Multiple Concrete Representations

The ALOE implementor can define multiple *views* of the abstract syntax through the use of several unparsing schemes in the grammatical description (see section 3.2.5). In this manner the implementor defines the *visibility* rules for his language. Different unparsing schemes can be defined to provide concrete representations at different levels of abstraction and detail.

Possible uses of multiple concrete representations in specific ALOEs include the following: at the top level the modular structure of a program can be shown, another unparsing scheme can be used to show all procedures and their specifications, another can show a procedure call cross reference, another can show the full syntactic expansion of the program and yet another can show a different syntax. For example, as shown in figure 2-7, in an $ALOE_{GC}$, a program can be unparsed with PASCAL syntax for those constructs that are equivalent in both languages [Feiler 82b]. Other examples of the use of multiple concrete representations in general are described in section 2.5.

Visibility rules can be coupled with the use of context windows. Figure 5-2 shows different context windows of a GANDALF data base. At the BOXES level, the names of all the modules are shown with the cursor placed at the module BasicOps. At the MODULE BasicOps level the state of the module with respect to project management, the specification of the facilities provided by the module and its versions are shown with the cursor placed at the

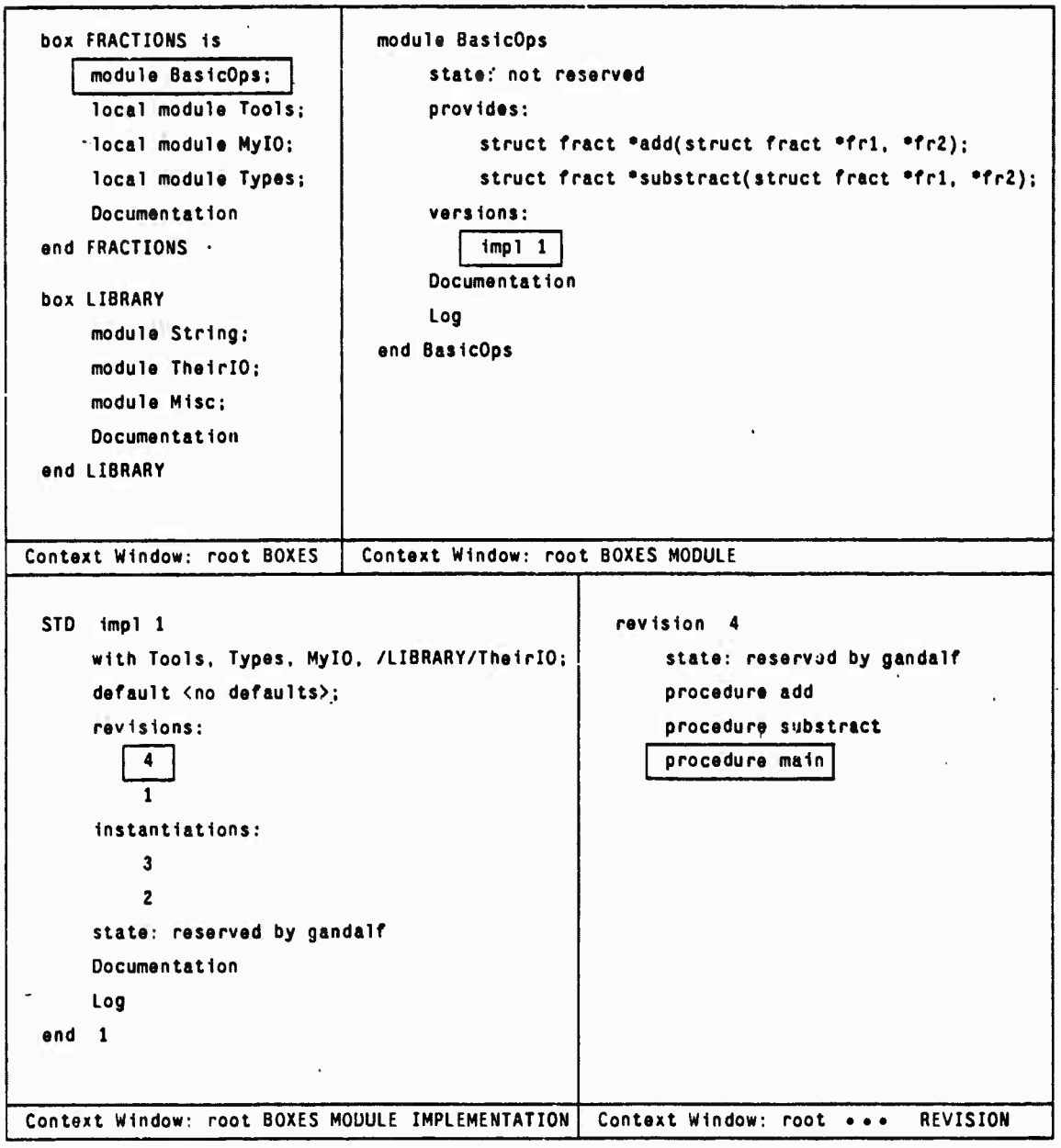

Figure 5-2: Context Windows in a GANDALF Environment

implementation '1'. At the IMPLEMENTATION 1 level, the specification of the implementation is shown with the cursor placed at the most recent revision. The REVISION 4 level shows all its procedures with the cursor placed at procedure main.

All these context windows are not shown concurrently on the screen as the figure may

seem to indicate, but they are overlaid in the same section of the screen (referred to as *program* window in section 2.6). Through cursor movement, these context windows are allocated and deallocated dynamically, and a one line *context stack* window shows the current nesting of contexts. With a better display technology, all these windows would not have to be necessarily overlaid. Every one of these windows, as well as all clipped windows, have their own cursor position. A possible extension to the basic capabilities of ALOE could include the ability to have more than one window (or stack of context windows) on the same tree or data base.

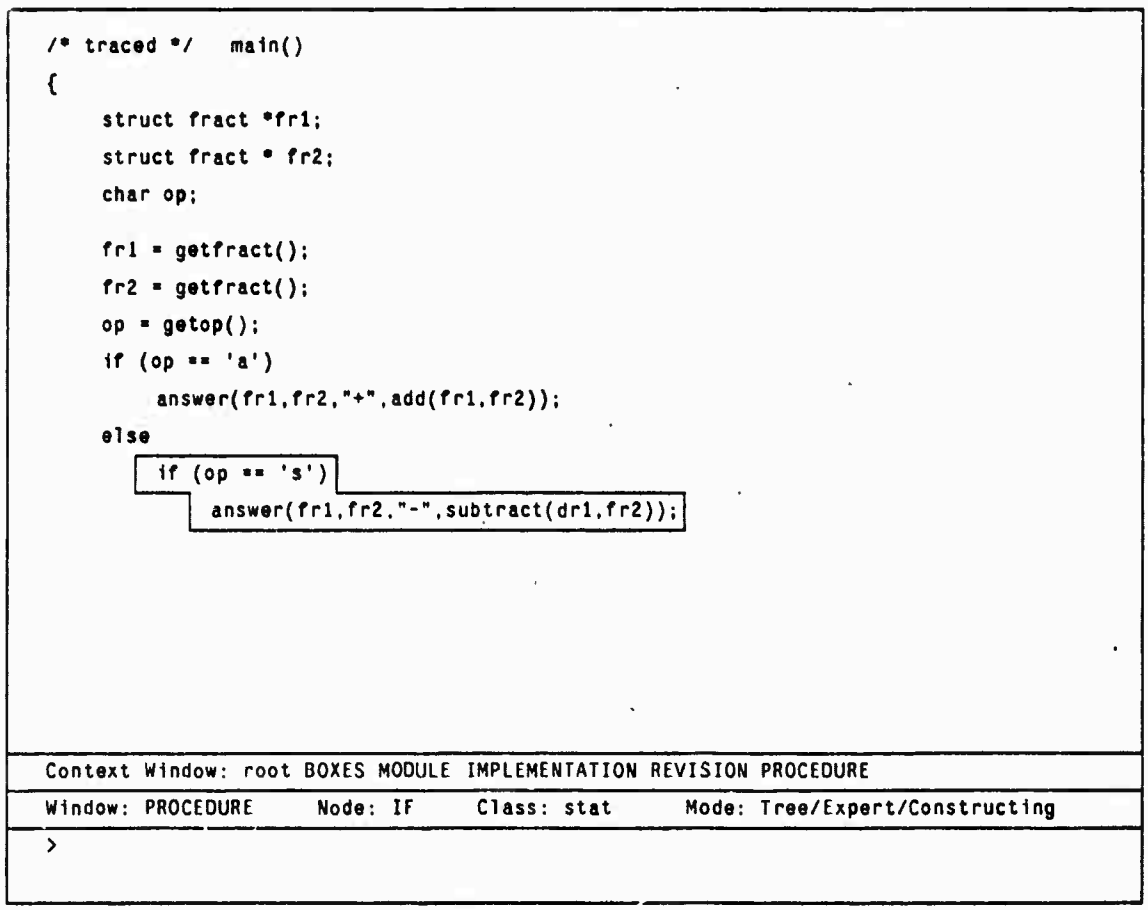

Figure 5-3: Procedure Context Window in a GANDALF Environment

Figure 5-3 shows a normal GANDALF screen with the PROCEDURE `main` context currently in the *program* window. The one line *context stack* and *status* windows and the *command* window are also shown.

In the GANDALF environment other types of information are also shown to the user. These include documentation that are pieces of text associated with the whole data base and with every module; log messages that keep track of the changes performed in the data base; debugging information that is provided through two independent windows: the *monitor* window used to show the values of certain requested variables as they change during program execution and the *callstack* window that shows the structure of the call stack at any point during program execution, for a detailed discussion of the representation of debugging information to the user, see [Feiler 82a].

There are cases in which an ALOE implementor may decide that certain nodes must be unparsed but the user must not modify them once they are created. The unparsing scheme language includes a special command that indicates that a subtree may be unparsed but it should not be visited. One example of its use in GANDALF is the date and user identification of a log message that are automatically generated and unparsed with the log message but that cannot be modified by a user.

5.3. Summary

The support provided by the ALOE system for the development of a large integrated environment, gives the ALOE implementor flexibility to *shape* the resulting environment. Through the implementation of action routines, the ALOE implementor decides when and how much semantic checking is performed, establishes the communication with other parts of the environment, *etc*. Through extended commands the ALOE implementor adds the necessary commands to provide the functionality needed for his particular environment. Through access control the ALOE implementor controls the behavior of the environment with respect to different kinds of users. With multiple unparsing schemes, the ALOE implementor provides multiple *views* of the environment and decides when and how much of the environment's data is shown at any particular time.

Chapter 6

Evaluation and Technical Issues

6.1. Introduction

In this chapter, an evaluation of the design decisions discussed in previous chapters, is presented, with an emphasis made on the technical issues involved. Some of the important aspects of the design of ALOE discussed and evaluated in this chapter include: its user interface, the effect that the equipment chosen has on the design, the features of programming languages that are particularly impacted in the design of a syntax-directed editor, the impact that building a generic system has on the design, *etc*.

A comparison is made with a text editing environment as well as with other syntax-directed editors, followed by a discussion of some strategies used in the design and implementation of the ALOE system and an evaluation of the success rate of the design decisions.

6.2. User Interface

6.2.1. Command Language Syntax

The command language syntax of an ALOE defines the communication language between users and environment. Some of the vocabulary of the language changes from ALOE to ALOE, because the language operators are part of the vocabulary: they are the constructive commands of an ALOE (see section 2.4.1). The basic editing commands remain constant but extended commands are specific to every ALOE.

The use of control character keys as synonyms for editing commands provides the necessary flexibility for a system with both novice and expert users. Novice users will use the

explicit name of a command (or its leading characters) and, as they gain experience, they will use the control character synonym. Cursor pad function keys are used to invoke cursor movement very successfully, other function keys could also have been used for the most common editing functions, as is done in the Synthesizer [Teitelbaum 81a].

Multiple language commands in one line saves intermediate states that could be considered unimportant and enhances the system response time by updating the display only after all commands have been applied. On the other hand, this could make it difficult for the user to recover from an error if the sequence of operators was not the desired one.

6.2.2. Editing Expressions

In most other syntax-directed editors and environments, programming language expressions are entered as text and parsed by the editor. Some systems [Teitelbaum 81a] are hybrid, that is, the rest of the program structure is entered constructively and expressions are entered as text. Other systems [Donzeau-Gouge 80, Archer 81b] simply support parsing for input of all the language constructs. Other systems [Alberga 81] even support incremental parsing for the whole program.

There is a clear motivation for parsing expressions: most users are used to text editing, parsing expressions is rather easy, and expressions are difficult to deal with structurally if one thinks of the hierarchical structure of expressions as specified in a BNF formalism [Backus 59]. Furthermore, expressions are displayed in infix form whereas to think of them structurally amounts to dealing with them in prefix (or postfix) form.

One of the research goals of this thesis was to investigate the feasibility of using structural editing at all levels of the language, including expressions. To this end, a different kind of grammatical description was developed (see chapter 3). An important aspect of its design is that it provides a way to express a *flat* structure for expressions instead of a hierarchical one. In this manner, all expressions are at the same syntactic level instead of in a hierarchy.

Experience with the use of ALOEs for several different languages showed that it was easier than anticipated to construct expressions. While unparsing, all expressions are automatically parenthesized by ALOE, using precedence values specified in the grammar (see section 3.2.4). The difficulties arise with editing the expressions. Special commands, such as **.nest** and

.transform (see figures 2-4, 6-2 and 2-5), that are not available in other systems (with the exception of the Synthesizer, which has been extended to support some kinds of transformations [Teitelbaum 82]), are very helpful for editing expressions.

The commands **.clip**, **.insert**, **.nest** and **.transform** are very successful because they are conceptually simple. The user has no difficulty in understanding the effects of the tree transformations performed by them. On the other hand, there are some transformations, such as a change from the expression `'a + b * c'` to the expression `'(a + b) * c'`, shown in figure 6-1, which implies a *simple* change of the evaluation order or the operations. The user may be used to thinking of this change as a simple insertion of parenthesis with a text editor. We could describe this transformation as a *rubber pull* in which a node of the tree is pulled upwards as if the edges of the tree were made of rubber. The change would be very difficult to express in terms of the modification of the tree structure, as can be seen from the syntax trees in the figure.

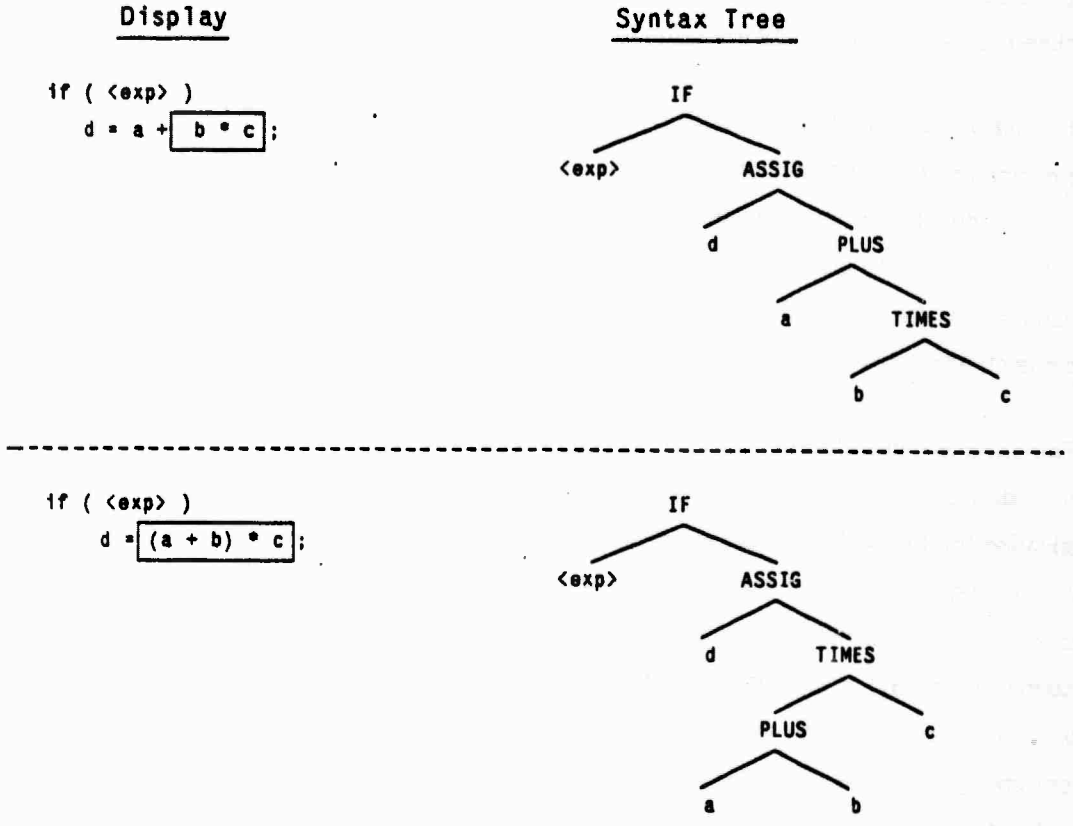

Figure 6-1: The Rubber-Pull Tree Transformation

A set of more complex tree transformation operations could be provided to solve the problem for the example above, and other similar ones. However, these complex operations may be difficult to understand and their effects difficult to anticipate. There could also be examples, such as a very large expression, for which the set of operations cannot be easily applied to achieve the desired transformation. The semantic implications are significant: an action routine call type (see section 4.3) would have to be added for every one of these transformation operations. The implementation of action routines would be made more difficult because code would have to be written for every one of these operations and would have to understand all the semantic implications of the transformation. On the other hand, these kinds of transformations would be very rarely used and so, their complexity is probably not cost efficient.

As the user can edit subexpressions, the user only needs to deal with those parts of the expression that must be modified. Any modification can be achieved with the use of the set of simple tree transformation operations of ALOE. As these changes are performed in a very localized context, the extent of the semantic effects due to the changes is very limited.

It could be said that the user should have the option to decide whether he wants to enter expressions textually or structurally. Indeed, it would be rather easy to provide an expression parser in the current ALOE system through action routines: a terminal operator of type constant is added to class *expression*; the user invokes that operator and gives the textual expansion of the expression as the constant's value. The action routine parses the expression and substitutes the node with the resulting subtree.

The argument in support of user choice could also be taken to imply that parsing should be provided at all levels of the language. The basic flaw with this argument is that the major motivation for structured editors is precisely that of dealing with programs structurally rather than textually. It is very confusing if the user has to enter his programs as text pieces and see and edit them as structures. There are some language-oriented editors such as the PDE system [Alberga 81] which support text editing for all constructs of the language. This system works in conjunction with an incremental parser that updates an internal parse tree after every change is made to the text. Section 6.8.4 gives a detailed comparison between PDE and ALOE.

Most users, if given the choice, will choose the method or tool they are familiar with

especially if they feel comfortable with it. This precludes them from investigating and learning the benefits of new tools or methods. This applies not only to the problem of editing expressions but applies in a more broader sense to the general problem of structure editing *vs.* text editing.

What makes expressions different from statements is that they are always shown in inflx form. There is a natural tendency to deal with statements structurally and this is not necessarily the case with expressions. Evidence of this situation is the hybrid design of the Synthesizer [Teitelbaum 81a].

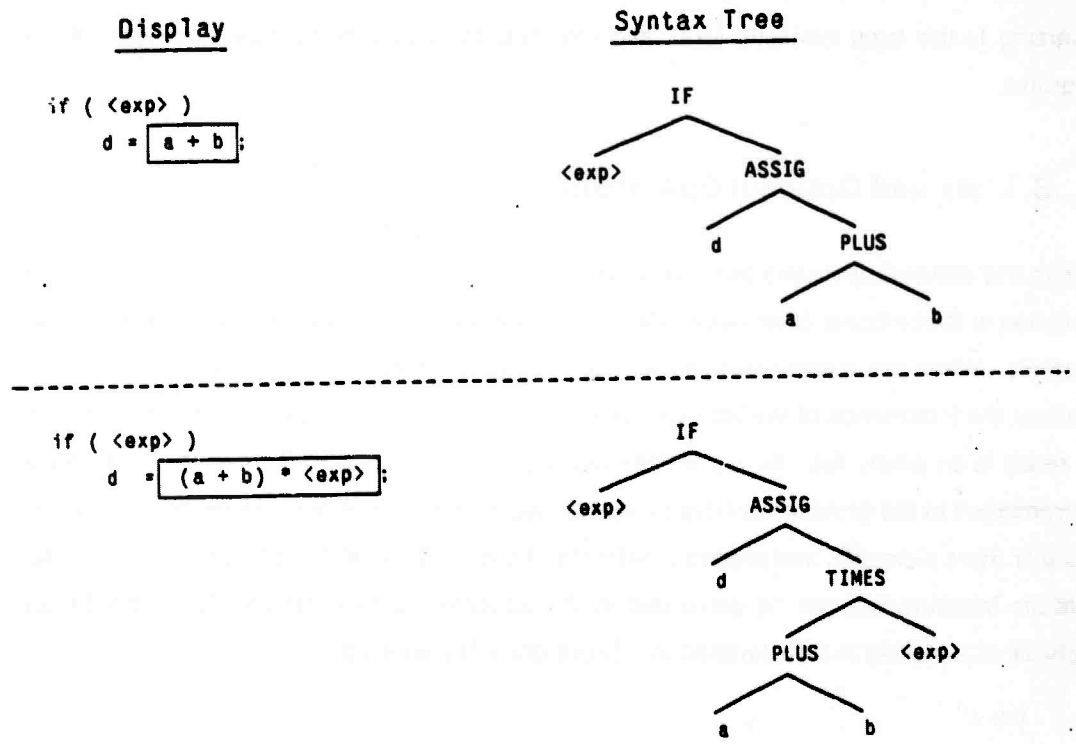

Figure 6-2: Nesting an addition into a multiplication

As we have already stated, one goal of a *friendly* user interface is to make it easy to accomplish simple and frequent actions. However, the fact that something was easy to do with some tool does not mean that the underlying concept is simple. For example, modifications that represent complex tree transformations can sometimes be accomplished in a simple manner with a text editor and might be much more difficult with a structure editor.

Other tree transformations like 'a + b' into '(a + b) * c' amounts to a simple .nest followed by the construction of 'c', as can be seen in figure 6-2. ALOE supplies the parenthesization of the resulting expression automatically. The user might have forgotten to include them if he was using a text editor.

It is very possible that the problem of dealing with expressions structurally can be solved by unparsing them in prefix form just as they are entered. Unfortunately this has two important drawbacks. First, users are too used to reading and understanding expressions in infix, even if they like to enter them to a structure editor or to a pocket calculator in prefix or postfix form. Secondly, we are generating editors for existing programming languages and would like to keep the syntactic correctness of the concrete representation of programs. However, this could also be solved by providing several unparsing schemes (see section 2.5), one of them unparsing to the *legal* syntactic form, and others to provide a better structural view of the programs.

6.2.3. Lists and Optional Operators

Lists and optional operators are related in several ways. This relationship can lead to some confusion in the behavior of an ALOE: All lists (*i.e.* variable arity non-terminals) in an ALOE can be empty. When the current meta node is an element of a list, a <cr>, used as a command, indicates the termination of the list expansion. If a meta node was the only element of the list, the result is an empty list. As we already discussed in section 3.2.2.2 there is a need for a differentiation in the grammatical description between lists of zero or more elements and lists of one or more elements, and probably with lists of two or more. If this difference is not made, syntactic inaccuracies can be generated in the concrete representation. For example, an empty list of variables in a declaration in C [Kernighan 78], such as

```
'int ;'
```

which is a syntax error: "missing variable name(s)". This can be solved in the current system by unparsing something special when the list is empty, but the syntactic error still exists. If this were a list of one or more elements, a meta node would be left in the list when the last element is deleted, so that it never becomes empty.

Another concept missing from the grammatical description is that of optionality. To solve this problem, an ALOE implementor would include the operator EMPTY in the classes where the operators are optional and the user could choose EMPTY as the operator whenever he

did not want to instantiate the corresponding optional operator. To improve the user interface, ALOE recognizes all operators that start with the characters EMPTY as special operators. Whenever <cr> is typed, and the current meta node is not in a list, EMPTY is automatically applied as a constructive command if there is an operator in the current class that starts with the characters EMPTY.

Consequently <cr> as a command has two meanings, that although are similar, could lead to some confusion. Empty lists and the EMPTY operator are two different structures. One is a non-terminal node and the other is a terminal one. Cursor movement and editing commands behave differently.

Another important problem with empty lists and EMPTY operators is their concrete representation. If nothing is unparsed for them and the cursor is moved into them, no highlighting is done and the user will be confused as to where the program cursor is. The information is included in the status window (see section 2.6), but that only tells the operator name and does not give its location. So, the user would know that the program cursor is at the EMPTY node but not where it is.

The unparsing scheme of a list can specify a special unparsing to be used when the list is empty. Similarly an EMPTY operator can be unparsed as a simple blank character. When the cursor is not at the node, its unparsing is virtually meaningless, but when the cursor is moved into the node, the single blank character is highlighted clearly indicating the cursor position.

Even with these facilities, dealing with optional operators can become very cumbersome in the cases of languages such as ADA [DoD 80] where there are many optional operators. One possible solution would be to design several types of optional operators: those that are very likely to be used, those that are not used frequently and those that are very rarely used. Unparsing could be done accordingly, the frequently used ones always shown whereas the rarely used ones only shown by special command.

Action routines can automatically make the nodes of rarely used optional operators into EMPTYs during the CREATE action call (see section 4.3.1) of the operator that creates the subtree where these optionals appear. This is also another useful application for multiple unparsing schemes: the normal unparsing scheme *hides* the undesired optionals, an extended command can be used to show them (by changing the unparsing scheme).

The Synthesizer [Teitelbaum 81a] has a special cursor movement command that lets the user go to an optional part whereas the normal cursor movement command skips these optional parts which are not even displayed if they have not been instantiated. The user must know that there is an optional part there. The proposal mentioned above, of using special unparsing and a specific extended command to show the optional parts is similar to the Synthesizer's solution.

6.3. Device Issues

6.3.1. General Characteristics

Display characteristics play an enormous role in interactive systems. Characteristics such as bandwidth, size of the screen (measured by the amount of information that can be displayed), ability to highlight parts of the screen, treatment of the screen as a two dimensional display rather than as a one dimensional scroller, input devices, *etc.*, can make a big difference in the usability of an interactive system independently of its *real* functionality.

For the purposes of our research we decided to investigate the feasibility of ALOE for a particular class of widely available terminals with a minimal set of hardware capabilities which included cursor addressing, insert and delete of characters and lines and highlighting capability. The initial implementation was built for the Concept-100 family of terminals [HDS 79], which has precisely these capabilities, and is being extended to other similar terminals.

If the bandwidth of the communication line were large enough to allow instant redisplay of the screen, ALOE would simply redisplay the whole screen after every interaction. As this is not the case even for the highest bandwidths normally available (*e.g.* 9600 baud) it is necessary to have an intelligent display interface that would update the screen optimally using the terminal capabilities to update only the parts of the screen that change after every interaction. To this end, ALOE uses the display package developed for UNIX[tm] EMACS [Gosling 81a, Gosling 81b]. The package uses an optimal updating algorithm.

The smaller the bandwidth the more important the display package becomes. As the bandwidth increases, the optimal algorithm becomes less critical. At a certain point the cost of redisplaying a whole line becomes cheaper (*i.e.* faster) than the sequence of character insert and deletes necessary to update it, because of the time it takes to perform the update

algorithm. Any redisplay algorithm should take this into account, the one ALOE uses [Gosling 81b] does.

As discussed in section 2.3, highlighting the area cursor is very important in the context of structure editors. The area cursor really makes the difference in emphasizing the structure of the program as the user moves through it. Highlighting this area cursor is a device characteristic that is critical if the area cursor concept is to have a real feedback and information value for the user.

The terminal highlighting capabilities will have an impact on the user interface of ALOE. The optimal situation is that highlighting be specified independently from the characters themselves, so that, in order to update a screen in which the only change from the previous one is the highlighting, the old highlighted characters and the new ones do not have to be redisplayed. This situation typically occurs on cursor movement which is one of the most frequent operations in an ALOE. The Concept-100 terminal has this capability.

The highlighting capabilities of most terminals require that the involved characters be redisplayed. This will work fine as long as the bandwidth is large. Some terminals have a rather unacceptable characteristic: for every line with highlighting two extra characters from the display are used to set the highlighting. This is a source of confusion to the user.

Finally, the type of highlighting is also important. As said before the purpose of highlighting is to show the area cursor, thus making a strong emphasis on the structure of the programs. Reverse video has worked just fine for these purposes. A blinking highlight would be distracting (end even annoying for large cursors).

6.3.2. Windows

The concepts of contexts and windows (discussed in section 2.6) are very successful in achieving the desired functionality. They also have proven their usefulness in other interactive systems suchs as the Display Oriented Interlisp [Teitelman 77], Smalltalk [Ingalls 78], and others. The implementation of windows is only possible when the screen is considered as a two dimensional display instead of as a one dimensional scroller (as in Mentor [Donzeau-Gouge 80]).

The size of the screen (typically 24 lines of 80 characters for the types of terminals

considered) is definitely a limiting factor in the usefulness of windows, not only because of the limited amount of space, but also because such windows cannot be easily separated (with clear borders) or partially overlaid (as we overlay pieces of paper on the top of a desk): borders would take away precious space needed for real information.

One of the first improvements would be to take advantage of terminals with larger screens or with additional local memory. Several screen images could be stored in the terminal's memory and then switching from one screen image to another would be instantaneous. This capability could be used to help solve the problems, discussed in section 2.6, related to the switching between different displayed contexts, the coexistence with user program I/O as well as to provide a larger window size for several of the windows mentioned in that section.

Beyond this, screens with raster scan displays are more suitable for windowing and distribution of different kinds of information [Teitelman 77, Sproull 79, Ball 80, Ball 81]. The use of these kinds of displays would greatly improve the user interface (and the usability) of ALOEs.

Nevertheless there are some limiting factors to the use of windows and contexts. First of all, given any display size, it would be used up rather quickly, and unfortunately, real screen size cannot be increased arbitrarily. On the other hand the amount of partially overlaid windows that the user can manipulate without getting confused is also limited. This means that, even with larger and better displays, screen organization and management for its optimal use is still an extremely important consideration in the design and implementation of any interactive system.

6.3.3. Wrapping Long Lines

There can be instances in which a line in a program is wider that the window used to display it. It is important to have some way of accessing those parts of the line that do not fit in the window. As a first approach to a solution ALOE scrolls the window to the left when the cursor includes a long line and tries to display the entire cursor. There are also explicit editing commands to scroll the window left and right. Unfortunately many times simple cursor moving commands in and out of those lines causes the screen to scroll back and forth which can be relatively annoying.

A solution to this problem, adopted by the Synthesizer [Teitelbaum 81a], is to move the starting point of a line to the left margin, if it would start past some predefined column position that varies according to the width of the window. The following lines would then be indented relative to this line. If a phrase (comments, expressions, *etc.*) is too long, it is broken and indented relative to its starting position in the previous line.

The solution to this problem is to have *flexible* or *conditional* line breaks, as proposed by Oppen [Oppen 79], incorporated to the unparsing schemes language. In this manner, lines are broken depending on the length of a particular line and the available width of the window.

6.3.4. Enhancements

As discussed in previous sections, even though the terminals we are dealing with have many advantages and are definitely sufficient for simple ALOEs, they are still very limited for the kinds of integrated environments that we want to develop, such as the GANDALF environment described in chapter 5. The number of windows is very limited and some of them must be overlaid. A better display could provide a larger collection of windows.

With respect to user input to ALOEs, the current system only supports keyboard input. Although the use of control characters and function keys has greatly improved the user interface, a pointing device, such as a *mouse* can be used to point to specific parts of the screen as in the Eravo text editor [Lampson 79]. Coupled with the display's improvements discussed above a pointing device would make possible the use of menus that have been used successfully in other interactive systems [Teitelman 77, Ingalls 78]. The pointing device could also be used to indicate cursor movement by moving it to different locations on the screen.

6.4. Language Features

In this section we try to identify those programming language features that are important in the ALOE context, that is, in the context of structured editing.

It is certainly the case that the language features that are well structured are best exploited by a structure editor. Whereas features that lack structure, such as macros and comments (see following sections) are definitely difficult to handle.

One interesting aspect of programming languages that is greatly impacted by structured editing is that of ambiguity of the language sentences. A typical example of ambiguities is the well known *dangling else* problem of certain languages. The ambiguity only exists in the concrete representation, because it has to be parsed and *else* parts have to be associated to the corresponding *if* statements. In an ALOE the user always indicates by *construction* the kind of statement he wants built (this is the principle behind the *constructive* approach of program building), and therefore, there is no ambiguity as to which *else* part matches every *if* statement.

As long as the text produced by ALOE's unparser will not be parsed, indentation can be used to indicate the matching *else* parts. In the extreme case of an application of an ALOE for which the text will be parsed elsewhere, action routines can generate dummy *else* clauses to solve the ambiguity of the concrete representation.

Another typical example involves constructs like `'fun(arg)'`. The ambiguity here is that in certain languages it could either be an array reference or a function call. Compilers must use semantic knowledge to disambiguate the construct. In an ALOE this is definitely not necessary because the user has already indicated the kind of construct he wants built. The concrete representation of the construct is totally unimportant except for feedback purposes.

Other types of *syntactic sugar* like punctuation marks such as commas, semicolons and parenthesis are often incorporated into languages to disambiguate constructs or to make the parser's job easier. As parsers are no longer needed, syntactic sugar can be much simplified and some of it even eliminated. New concrete representation styles can be adopted that give the user better feedback about the real structure of his program but that do not have to be parsed and neither has the user to type them.

This also affects language design because it is now easy to deal with ambiguous languages, it is no longer necessarily *bad* feature for a language to have ambiguous constructs (with respect to parsing) as long as their visual representation is unambiguous (by, for example, assigning meaning to indentation level).

6.4.1. Macros

The idea of macros and structured editing clash in some important ways. Macros conceived just as text substitution have no *structure* to them. Macros are designed with a preprocessor in mind: the preprocessor performs the textual substitution before the parser processes the program. The program that the user sees is different (in its form) from the one the compiler gets. In the context of structured editing there is no such thing as *text substitution*.

Most uses of macros in languages such as C [Kernighan 78] are due to lack of richness of the language itself. Language constructs such as constant definitions, in-line procedures, type definitions and enumerated types, *etc.*, should be used instead of macros (version 7 of C fixed the last two items in this list). Some other uses include modification or abbreviation of language syntax, and these can be solved through different concrete representations through unparsing schemes.

The design of the ADA programming language [DoD 80] solves the problem of macros by providing some of the language constructs mentioned above as well as *generics*, some instances of which can be thought of as some kind of *structured macros*.

6.4.2. Comments

Comments are extremely important because they are the *only* way to provide *in-code* documentation, which is fundamental for good programming style and practice. The problem with comments is that they have no internal structure. They are mostly designed with lexical conventions and designed to be thrown away by the lexical analyzer and not kept or processed by any other tools. Comments are normally not considered to be part of the syntax of a language, or a meaningful language construct.

A comprehensive design of structured comments can be added to any language incorporating them into the abstract syntax. This design would differentiate between different types of comments and their concrete representation would use different formatting styles for them, and would, therefore, make it much easier to understand documentation.

ALOE itself does not understand the concept of comments (as it understands the concept of optionality through the EMPTY construct, see section 6.2.3). Comments are incorporated into

ALOEs directly through the grammar as string constants. As part of the design mentioned above, ALOE could understand comments as a special concept in itself.

One of the biggest problem with comments is that even though in the context of ALOEs, parsers are no longer needed, there is sometimes the need to write a parser to translate programs into ALOE's internal language representation to be able to incorporate existing textual programs into the environment. The parser must then understand comments and must be able to decide where and how to associate the comments with the internal structure. As there are no fixed rules as to how comments are associated with the language constructs and every user has his own conventions, the task of the parser is close to impossible. There will always be situations in which the comment will be associated with the wrong construct. For example, some people like to put a comment before the statement it is referring to and others like to put it after the statement. In many situations the reader must understand the comment to be able to determine which construct it is referring to.

Comments are very necessary in programming languages but they should not be unstructured. New programming language design should incorporate comments in a structured way associating them with the construct of the language. The PL/CS language of the Synthesizer [Teitelbaum 81a] includes commented statements. These comments are also used as placeholders for the whole statement for elision purposes (to hide implementation detail to be able to show a greater context in the screen). The structured comments design should not be too simplistic or restrictive so that the user will not feel that he cannot place all the comments he would like to include.

6.4.3. Extensible Languages

ALOEs have difficulties with extensible languages: the syntax tables, generated by the ALOE generator, are static, and thus new operators cannot be added. This precludes the possibility of making new operators or types become legal operators of the language. For example, it would be desirable to add every new type as a legal operator of class *type*. The problem is that the syntax tables contain the *permanent* structure of the language and information in them is used when files are read or written. The *temporary* operators would have to be kept in a separate table where information is not used for storage but only for user interface.

6.4.4. ALOE for Other Structures

Throughout this dissertation we have emphasized the generation of ALOEs for programming languages and systems because they are their motivating application. ALOEs can also be generated for other languages and structures that can be expressed using the grammatical description. An important example is *Aloegen*, the ALOE for creating and editing grammatical descriptions. The syntax of the grammatical description can be expressed in terms of itself and thus, the same generation process can be followed as for other ALOEs. Other interesting examples include an ALOE for SCRIBE [Notkin 82d] and an ALOE for a mail system [Notkin 82a].

SCRIBE [Reid 80] is a document production system. SCRIBE documents are very structured. Most of them are divided into chapters, sections, subsections, *etc*. Within these structures there are other structures that specify special formatting characteristics such as tables, examples, itemized lists, *etc*. These structures can be expressed using the grammatical description of the ALOE generator. The terminal generators of such an ALOE are paragraphs of text.

An electronic mail message has a fixed structure. It has a header and a body. The header is divided into several fields, such as the identification of the message, the sender, the recipients, the subject of the message, *etc*. This structure can be easily described using the grammatical description and an ALOE can be generated for it. Invocation of the mailer system, once the message is composed and the reading of messages from the mail boxes of users, would be done through action routines and extended commands.

Before any realistic use of ALOEs for these applications can be done, it will be necessary to add support for direct text editing within ALOE instead of invoking a separate text editor for large pieces of text.

6.5. Language Issues

6.5.1. Editing language vs. Edited Language

Traditional programming environments provide a collection of languages. Every tool of the environment has its own language, so the overall language is an ill-formed one with no consistency or uniformity. The user must remember which tool of the environment he is communicating with.

By providing a uniform user interface to an integrated environment through ALOE, the language of communication is the command language of ALOE. Part of the vocabulary of this language changes from ALOE to ALOE: the operators of the language are the constructive commands of ALOE (see section 2.4.1).

It is important to note the difference between this editing language and the language of program or structures being edited (the *edited* language). ALOE does not use this edited language as its language of communication, but, on the other hand, there is an important connection between the two languages: the constructive commands of the *editing* language (*i.e.* part of its vocabulary) represent the constructs of the *edited* language.

An extension of the capabilities of the ALOE system could include the possibility of editing the *editing* language, so that editing macros could be provided.

6.5.2. The Grammatical Description

The ALOE grammatical description is *operator* oriented, that is, the emphasis is placed in the constructs of the language rather than on a collection of productions. When the ALOE implementor assigns names to the constructs of his language in the grammatical description, he is actually defining part of the vocabulary of the environment language (the *editing* language).

The separation of abstract syntax and concrete representation, which allows the specification of multiple concrete representations, permits variants in the form of the edited language. We can think of them as different *views* of the language.

As we have already pointed out, some important concepts are missing from the design of the grammatical description. It is lacking lists with at least one element (*i.e.* lists that cannot be empty), it should have a way of expressing optionality and one should be able to include associativity values in addition to precedence values for operators.

Non-empty lists are necessary to be able to avoid syntactic inaccuracies in certain constructs of languages that require them (see example in section 6.2.3). Optionality is a concept that appears in most languages and it would be much better if ALOE would understand the concept rather than include it by way of the special treatment of operators that start with the characters EMPTY (see section 6.2.3). Associativity values are needed for the correct parenthesization of expressions that include non-associative operators, for example the expressions `'a - b - c'` and `'a - (b - c)'` are not equivalent.

All of these missing concepts should be included in the next implementation. The problems caused (as already discussed) are well understood and their solution is feasible. They were not added to the current version because other additions and improvements were considered more important and not because any important technical difficulties.

6.5.3. Unparsing Schemes

The experience with the ALOEs generated so far tells us that, when the formatting of programs is done automatically and in a *reasonable* and consistent way, the users rapidly get to like it. The main reason for this is that users do not have to deal with formatting explicitly any more. It is also the case that several *formatting styles* can be incorporated into any ALOE through multiple unparsing schemes.

In sections 2.5 and 5.2.8 we included many examples of the different uses of multiple concrete representations. Unparsing schemes provide a very powerful mechanism to achieve the translation from the abstract syntax structure into several concrete representations. The ability to change from within an unparsing scheme, the scheme used to unparse the offspring of a node provides some ability to unparse based on context. For example, in the GC [Feiler 79] to PASCAL [Jensen 74] translator [Feiler 82b], referred to in the example of figure 2-7, the GC increment construct `'i++'` is unparsed in the PASCAL version as `'i := i + 1'` except when it is located as the increment in a `for` loop, in which case it is unparsed as the keyword `TO`. Similarly, the decrement construct `'i--'` would be unparsed as `DOWNTO` if found in the `for` loop.

However, the unparsing scheme language does not provide by itself all the desired functionality of a general unparsing mechanism. The basic problem is that the concrete representations are fixed because they are statically determined by the unparsing schemes. Some dynamic elements can be obtained if the unparsing schemes are combined with action routines.

An example of such an occurrence, taken from the GC to PASCAL translator [Feiler 82b], is the `return` statement in a function. In GC, the statement is unparsed with the keyword `return` followed by the value returned. In PASCAL the value is assigned to the function name in the body of the function. When the statement is being unparsed, the unparsing scheme has no access to the name of the function at all, and so, direct translation cannot be accomplished. Action routines can be used to achieve the desired effect in the following way: the structure of the return statement is defined to have two offspring instead of one. The action routine associated with the `return` operator will fill the name of the function. In the unparsing scheme for GC only the value returned is shown. In the PASCAL scheme the name of the function is also unparsed and is available at time of unparsing.

Another similar example occurs in the implementation of *Aloegen* [Notkin 82c], the ALOE used to create and edit grammatical descriptions. The main unparsing scheme is used to show the structure of the grammar as is shown in the example of figure 3-1. The other unparsing schemes are used to produce the syntax tables that form the language knowledge of an ALOE. At some point in the generation of these tables it is necessary to know the sizes of some lists. This knowledge is not available directly from the unparsing schemes. This problem is solved in a similar way as the previous example: an additional offspring is defined for such constructs, its value is filled in by the action routine, it is unparsed when needed and ignored by the other unparsing schemes.

The solutions to the problems mentioned in these two examples use the basic mechanisms of ALOES: action routines and multiple unparsing schemes. However, they require the redefinition of the abstract syntax structure to achieve the desired result, even though the *logical* structure of the language is not changed. It is the lack of processing power at unparsing time what forces these modifications. In section 3.2.1 we discussed the difficulties associated with the modifications to the structure of operators in the grammatical description.

In order to provide a more general mechanism, several additions to the unparsing scheme

language must be made. These include control structures, such as conditional unparsing schemes that could test several conditions and unparse based on them, loop constructs for unparsing elements of lists, and functions specified in the unparsing schemes and invoked by the unparser, that would compute the necessary information without requiring a change in the structure of the operators. Other simple extensions would include mechanisms to deal intelligently with long lines and the ability to specify conditional line breaks, as was already mentioned in section 6.3.3.

6.6. Generic Systems

It is different to build a syntax-directed editor for a particular language than to build a generator of such editors. Some design decisions are influenced by this difference. In particular, design decisions must be made to provide solutions with general mechanisms rather than providing a specific solution that solves a problem for one language but not for another.

For example, in the design of the command language for an editor built for a particular language, such as the Synthesizer [Teitelbaum 81a], the command names for editing and language commands can be chosen so that they would not have similar leading characters in their names. This allows a single naming convention to be used for both kinds of commands. In a generic system, the names of language commands are not known in advance, they are determined by the language description. The choice of different naming conventions for both types of commands lets the ALOE implementor select good mnemonic names for his language constructs without having to be concerned with the names of editing commands. It may also be desirable to make the user aware of the difference between editing and language commands. Introduction of new editing commands will not cause the ALOE implementor to rename some of his language commands that could have similar leading characters in their names as those of the added editing commands.

Another example is the choice of cursor display. A single character cursor may be ambiguous in some instances (as seen in figure 6-3). In a particular language there may not be any situations in which a single character cursor is ambiguous, or there may be just one language construct that is ambiguous (as in the case of labeled statements in PL/CS). If an editor is built for these languages, the single character cursor may be a good design decision, but in a generic system it is important to provide a more general mechanism, such as the area-cursor of ALOE, that solves the problem of these potential ambiguities.

In an editor for a particular language, unparsing rules are part of the code of the implementation. In a generic system, these rules must be specified in some language. The unparsing scheme language of ALOE described in section 3.2.5 is one such language.

6.7. Comparison With Text Editing Environments

It would be unfair to compare structure and text editors for creating and editing programs, in terms of the time it takes to do it. The aims and goals of both types of systems are very different. For the text editor the contents of the file being edited is unimportant, it only understands about characters (and possibly lines and screens). No knowledge or support (except for some small support for formatting programs and template expansion in some editors) is given when the text editor is used for editing programs as opposed to editing a document. ALOE is knowledgeable of the contents of the entity being edited. The ALOE structure allows the environments to take advantage of a large collection of information for processing while the user is entering and modifying his program. This means that an ALOE may be doing much more processing than just ensuring syntactic correctness. This can include semantic checking, automatic generation of program pieces, lexical analysis, invocation of the code generator, *etc*. (see section 4.2).

The compilation cycle is a more realistic unit for comparison (for aloes for programming languages). It can be defined as the time (or number of keystrokes) it takes to construct a program and get it to compile correctly in the syntactic sense (*i.e.* no syntax error left). This often requires successive invocations of the text editor and the parser.

It is also true that the text editor and the corresponding parser were not designed as an environment, so we would be using two almost unrelated systems together, for comparison against an integrated environment. This means that the two systems are not directly comparable in terms of time or keystroke count.

On the other hand, the combination of UNIX[tm] EMACS [Gosling 81a] and the *Make* facility [Feldman 79] in UNIX[tm] can form an interesting compiling environment in which UNIX[tm] EMACS understands the error messages caused by the compiler invocation (through line numbers associated with these errors), and places the user in the position in the text file where the error occurred. This is of great help in reducing the time it takes to get a program correctly compiled. But, of course, UNIX[tm] EMACS is not preventing any kind of errors nor is it

doing any extra processing while the user is editing his program, except for some support for formatting programs of certain languages that it knows about.

ALOE guarantees the syntactic correctness at all times. This correctness is with respect to the abstract syntax structure. A concrete representation can be provided that would be syntactically correct if the program text were to be parsed, even though it may not be the *main* concrete representation preferred by the user. As previously discussed, ALOE may also be doing other kinds of processing. Formatting is also handled automatically and several formatting styles can be supported by an ALOE. These styles are expressed in terms of unparsing schemes and are not modifiable by the user. A possible extension of ALOE could allow the user to define unparsing schemes as long as he does not get more access to nodes than what the defined unparsing schemes give him.

Another comparison measure is the size of the program files stored as text by a text editor and stored as trees by ALOE. ALOE stores semantic information (currently one computer word per node in the tree) that can be reused when the program tree is read. If this were not the case, the size of program tree files would be smaller that the corresponding text files. With the added semantics and some space used as a file header to identify the language of the ALOE and other information, the sizes of program files are comparable for large programs (*i.e.* of at least 60 lines of code), while for smaller programs the text files can be of anywhere from half the size to comparable size of the corresponding tree files depending on the amount of white space (*i.e.* indentation) contained in the text file. The reason for this reasonable file sizes is that all the keywords, punctuation marks and all the white space required in the text file are not stored in the tree file.

6.8. Comparison With Other Syntax-Directed Editors

In this section we will discuss the similarities and differences of ALOE with other syntax-directed editors. We do not intend to discuss all of these editors, but we have picked a set of well known editors for this comparison.

6.8.1. The Cornell Program Synthesizer

The Cornell Program Synthesizer [Teitelbaum 81a] goals are quite different from ALOE's. This explains the different design decisions. First and most importantly, ALOE is a generic system, that is, it is a generator of syntax-directed editors instead of an editor built "by hand" for a particular language. The Synthesizer is a syntax-directed editor implemented for PL/CS [Conway 76] a small subset of PL/I. The Synthesizer does not use different naming conventions for editing and language commands. Editing command names were designed so that they wouldn't conflict with the PL/CS language commands.

The Synthesizer was implemented for a microcomputer and for this reason its power has to be limited to work for small programs. ALOE was designed to permit the generation of large integrated software development environments such as GANDALF [Habermann 79b, Habermann 82], described in chapter 5.

The Synthesizer is a hybrid editor that combines a structure editor for the high level language constructs (*e.g.* declarations and statements) and text editor for low level constructs, called *phrases* (*e.g.* expressions, assignments and parameter lists). This design has an important impact on the user interface. Cursor movement is different depending on where in the program the cursor is. At a high level structure it moves structurally, at phrases it can move character by character.

Construction commands also have this difference. At the high level, they are commands to introduce program templates, at phrases they are plain text. This implies that after the input of a phrase and after editing it, the phrase must be parsed into the internal structure. Information about parenthesization must be kept in the internal structure so as to be able to reproduce the phrase as the user typed it.

ALOE provides a uniform interface at all levels of the program structure. There is no need to parse these expressions, and subexpressions can be handled separately. The Synthesizer cannot treat subexpressions as structures in themselves. In particular, a subexpression cannot be pointed at and extracted from an expression and inserted elsewhere. Section 6.2.2 contains an explicit discussion of the differences between text and structure editing of expressions.

The program cursor in the Synthesizer is a single character cursor as opposed to the area

cursor of ALOE. This causes some ambiguities in cases such as the one in figure 6-3 where the cursor is at the `label` and it is not clear if it refers to the label only or to the whole labeled statement. The area cursor of ALOE solves this ambiguity and also makes more emphasis on the structure of the program. In ALOE the cursor can be placed at a list, that is, including all elements of the list. In the Synthesizer the cursor is only placed at single elements of lists, in order to apply editing commands to lists, the first and last element of the list must be specified. A single character cursor would be ambiguous if the cursor were allowed to point to the whole list. A possible advantage of the single character cursor is that, if used to trace program execution, the cursor is changing rapidly and a single character cursor may be better in this case than the area cursor [Teitelbaum 81b].

```
label:   IF (K > 0)
         THEN statement
         ELSE statement
```

Figure 6-3: Single character cursor in the Synthesizer

The Synthesizer is not only a syntax-directed editor, it is a programming environment that integrates interpretation and debugging facilities. Its characteristics as a programming environment should be compared with LOIPE [Feiler 82a, Medina-Mora 81b] an integrated programming environment based on ALOE.

Current developments of the Synthesizer include the development of a generator of Synthesizer-like editors with the use of attribute grammars for expressing the semantics of language templates [Reps 82]. Section 4.8 contains a discussion of the use of attribute grammars in syntax-directed editors.

6.8.2. The MENTOR System

The MENTOR System [Donzeau-Gouge 80] is a structured editor for PASCAL [Jensen 74]. It also has goals that are different from ALOE's. MENTOR is based on parsing of input for all levels of the language although it also supports some form of constructive editing. The MENTOR implementors decided to support parsing of input because users were more comfortable with writing their programs as text from their experience with text editors.

The problem with parsing on input is that users enter their programs as text but then they

must deal with them structurally for editing and inspection. It can be very confusing if the user enters his program as text, but must edit it as structure.

MENTOR has a scroller type display interface, similar to that of some LISP systems [Teitelman 78, Perdue 74] instead of the two dimensional display interface of ALOE. This means that after every interaction, if the user wants to see the result of his action, the current display is not used and the subtree on which the user is focused must be redisplayed again.

MENTOR uses MENTOL, a tree manipulation language, as its command language. MENTOL is used for all manipulations from cursor movement to searches using pattern matching to tree transformation and context sensitive checking. Cursor movement can be particularly cumbersome because, to get feedback on the result of the cursor movement, the subtree must be redisplayed again instead of the immediate feedback given by ALOE which only changes the highlighting in the screen.

Complex pattern matching structures can be handled in MENTOL; to perform searches, for example, the command

```
@TXT F @ if $V1 then X:=$V2 else $V3
```

will look in the subtree denoted by the marker @TXT for the next IF statement containing an assignment to variable X as its *then* part. This seems to be a rather complicated way of achieving the desired search. The .find command in ALOE, described in section 2.4.2.1, provides better feedback by automatically displaying the result of the search. It is much easier to invoke, and it is incremental so that, if the first attempt at locating the desired node fails, the command can be repeatedly invoked without having to specify the search parameter again.

The MENTOR user can control the depth to which a subtree is displayed. This concept is referred to as *holophrasting* in [Hansen 71]. As MENTOR does not have multiple concrete representations and, since its display interface is not interactive, it must redisplay the subtrees every time the user wants to see them, the concept of controlled depth is fundamental in MENTOR. In this manner, larger contexts can be visualized and it also takes less time to display a particular subtree. Unfortunately, simple depth restrictions do not necessarily convey the desired level of abstraction in languages such as C [Kernighan 78].

For example, certain depth level may show all the details of a declaration including its initialization and none of a nested statement. Constructs at the same *conceptual* level some times do not appear at the same depth level.

Another problem of this type of interface is that the context around a particular construct is not shown when the cursor is at the construct. The user must move the cursor up to some node in the tree and redisplay again if he wants to see the surrounding context. ALOE will always place the area cursor in the middle of the window that is currently being used for display of the program. In this manner the surrounding context is always visible. Different abstraction levels can be achieved through multiple unparsing schemes that also are knowledgeable of the structure of the language, and will show constructs at different depths but the same contextual level.

MENTOR is not an integrated programming environment. Its internal representation is not used by a compiler to generate code, nor is this step automatically invoked as is the case in LOIPE [Feiler 82a, Medina-Mora 81b] or GANDALF [Habermann 82, Notkin 82b] (see chapter 5). The MENTOR user must explicitly unparse his tree into a text file and give it to a compiler for processing. The compiler, of course, will duplicate many of MENTOR actions. There is no direct support for program execution, only for program building.

MENTOL is used to build routines to check context sensitive properties of PASCAL programs. MENTOL is not a general purpose programming language but a tree manipulation language with the ability to manipulate attributes in nodes. The ALOE implementor through the use of action routines has all the support of the ALOE implementation environment, described in section 4.6, to write the equivalent routines in C [Kernighan 78].

6.8.3. The Emily System

The Emily System [Hansen 71] was one of the first efforts with syntax-directed editing. It is a menu-driven system, with selections made with a light pen pointing at a graphics display. The programmer constructs a program by selecting a BNF production to replace the current non-terminal node. The BNF productions in Emily include the concrete representation for the constructs (keywords, separators, terminators, *etc.*), and so, the programmer still had to be aware of the details of the concrete syntax.

Emily was not very fast (the major delay was in filling the screen after every interaction), and experience with it showed that it took longer to construct a program using Emily than with a text editor, even though there was a considerable saving of keystrokes. It should be noted, however, that fewer errors were made when using Emily. Simple editing operations, such as the deletion of a construct, required several interactions, instead of one, thus making the editing of program somewhat difficult. Emily was not an integrated system, in order to compile a program, text was produced an had to be parsed, the internal structure created by Emily was not used.

6.8.4. The PDE System

The PDE1L system [Mikelsons 80] is a program development environment for PL1L, an extended subset of PL/I. It is a successor of LISPEDIT [Alberga 81], an environment for LISP/370. PDE1L is not a syntax-directed editor, but it is language-oriented, that is, the interactions with the system are based on the language constructs, but the user manipulates the textual representation of the program. PDE1L is an integrated environment that incorporates to the editor an incremental parser, an interpreter, a compiler and a debugger. As in ALOE, the user interface of the editor provides a uniform user interface to the environment.

By manipulating the textual of his program, a user can introduce syntax errors to his program. For example, the user realizes that there is a missing END statement because the indentation of his program is not what he expects. Syntax errors, when detected, are displayed in two different forms. When text that is inserted in the program cannot be parsed consistently with the surrounding text, it is highlighted to indicate so. When text is deleted in such a way that the remaining text cannot be parsed, the deleted text is replaced with one or more meta symbols that indicate the nature of the missing material. In ALOE, syntax errors simply cannot occur.

PDE1L has a large set of editing commands that are applied to the current *focus*, the equivalent of the cursor in ALOE. The focus represents a portion of the parse tree. The focus can be changed through cursor motion commands. The user is then dealing with his program structurally, but he still has to enter it as text. The user has to concentrate both in the concrete form and in the content of his program. In ALOE, the user always deals with his program structurally and can concentrate in its content without worrying about its concrete form.

PDE1L has a very sophisticated display algorithm [Mikelsons 81] that considers the relative importance of the language constructs with respect to the *focus*, when deciding which constructs are unparsed and which are elided. One problem with the algorithm is that the relevant parts of the parse tree are traversed twice to achieve the correct unparsing.

In PDE1L, all input is entered in a special input section of the screen and not directly where it will be inserted. When the user finishes the input it is parsed and inserted in the correct place. Feedback on errors is not given until the text has been parsed and inserted and not as it is typed.

6.8.5. The Interlisp System

The Interlisp System [Teitelman 78] is a very sophisticated programming system for LISP. The simple syntax and semantics of LISP lend themselves very well to more structured manipulation of programs, its interpretive nature lends itself better to the edit/interpret approach. Interlisp incorporates powerful facilities like structured editing, sophisticated debugging techniques, automatic error correction, the analysis subsystem, the programmer's assistant and others. Many of the ideas present today in syntax-directed editors were first introduced in LISP environments. The display-oriented programmer assistant [Teitelman 77] makes excellent use of sophisticated display and window manipulation mechanisms [Sproull 79].

Interlisp does not provide a uniform user interface. There is a different interface depending on the tool of the environment the user is communicating with: the editor, the debugger, the analysis subsystem, *etc*. Different interfaces for different tools focus the user's attention on the tool rather than on the program being developed.

Input to Interlisp is given as text and it is parsed and inserted at the current position. Editing is done structurally. This means that the user has to deal with both text and structure when he is editing his program. This is not so important for LISP systems because of its very simple syntax, although syntax errors can be made and feedback from the system is only given after all the input has been processed and not as it is typed.

6.9. Design and Implementation Strategy

This section discusses a set of strategies used in the design and implementation of ALOE qualified by the experience gained through the actual implementation.

6.9.1. Frequent Operations

As discussed in chapter 2, one of the goals of an editor like ALOE is that, frequent operations, such as cursor movement and simple constructive and editing commands, should be easy to provide and perform. They should also be as efficient as possible so as not to cause a significant delay in the response time of the system.

One identifiable problem in ALOE in this respect is the input of variables, a very frequent operation in any language oriented editor. In ALOE, the name of the operator (*e.g.* IDENT) or its synonym (*e.g.* a quote mark (')) must be entered before the variable name, which is entered in the command line or as an answer to an explicit prompt. The suggestion to add lexical specification or lexical routines described in section 2.4.1.2, would solve the problem of having to type a command or a synonym for terminal operators. If the user makes a typing error, the variable must be deleted and entered again. This is an implementation flaw in the current ALOE that will be fixed in future versions, by providing text editing capabilities for terminal operators. This problem does not occur in systems such as the Synthesizer [Teitelbaum 81a] in which expressions (which contain the variable names) are entered directly as text and edited with text editing commands.

6.9.2. Infrequent Operations

There are some *attractive* features from the theoretical point of view that could be added to a syntax-directed editor. It could be very difficult to implement them, and would probably be used very rarely, and thus would not impact in any significant manner the performance and usability of the editor. Examples of these features include complex tree transformations that could be useful in a very limited set of cases and which will not be applied frequently. On the other hand, experience with these transformations would help understand and experiment with new ways of interaction with structures.

For example, the tree transformation that would change the expression 'a + b * c' into

'(a + b) * c' (see figure 6-1), which could be thought of as a simple parenthesization operation, involves a complex tree transformation as can be seen from the figure.

On the other hand, the user would probably use sequences of **.clip**, **.insert**, **.nest**, **.transform**, **.delete** and constructive commands to achieve the same results before realizing that the desired transformation is actually one of the provided ones. One explanation of this situation is that the transformation performed by these commands are simple transformations and it is easier for the user to identify their need and their usefulness that it would be with the more complex tree transformations. We have said before that the **.nest** and .(transform) commands, described in section 2.4.2.2, were very useful commands. It turns out that, they are really not used that often, because the situations in which they can be applied do not occur very frequently.

It should also be noted that the impact of **.transform** in the action routine interface is a significant one. As discussed in section 4.3.4, the action routines implementation has to take into account all possible semantic effects that such a transformation implies. More complex tree transformations would only make this task more difficult.

6.10. Conclusions

6.10.1. Successful Aspects

In summary, the aspects of the design of ALOE that have been more successful are:

- Separation of abstract syntax and concrete representation, which permits the structure of programs, not their form, to be emphasized.
- Multiple concrete representations, which permits different *views* on the same data, and also different contexts and levels of abstraction.
- Uniform user interface for large integrated environments, which allows the user to deal with his environment as single system instead of as a collection of unrelated pieces.
- Area cursor which permits to place the emphasis on the program structure.
- Synonyms for editing and language commands, that provide the necessary flexibility for novice and expert users and help in constructing and editing expressions.

- Efficient use of the limited screen space, as a two dimensional display, given the restrictions imposed by the choice of terminal.
- Support for large integrated environments, which makes it possible to automatically invoke other tools of the environment.
- Data base partitioning which makes possible the definition of different contexts in large integrated environments.
- Flexibility for the ALOE implementor to *shape* the behavior of the environment.

6.10.2. Missing Features

In different places in this dissertation, we have pointed out some of the important missing features in the design of ALOE, and at the same time, solutions to these problems have been proposed.

- Underlying text editing capabilities, fundamental for editing variables and constants and for development of ALOEs for structures other than programming languages.
- Lexical knowledge, which is extremely important for a better interface for constructing and editing expressions.
- Complex tree transformations (although we have argued that their inclusion might not be cost efficient).
- Some form of an undo command, which is very necessary for a smooth recovery when errors are made.
- Non-empty lists and optional operators in the grammatical description.
- Control constructs and function invocation in the unparsing scheme language.

6.10.3 Further Research

Solution to these missing features would constitute a natural extension of this work. Other areas that could also be considered for future research include:

- User interface. More work and experimentation is necessary to fully understand the important aspects and fine points of the user interface of syntax-directed editors.
- For context sensitive processing, some form of synthesis of action routines - that provide a flexible mechanism but little control - and attribute grammars - with some modifications to increase their efficiency - seems worth investigating.

- Important aspects of language design can be influenced by the existence and use of syntax-directed editors.

- We have succeeded in producing a generator of extensible syntax-directed editors that support the development of large integrated environments. The next step is to produce a generator of such environments, that would automate more aspects of their development.

Appendix A

Editing Commands

Editing commands are common to all ALOEs. These commands are invoked by typing a dot (".") followed by the name of the command and a carriage return. Only enough characters to designate the command unambiguously need be entered. For the common commands, one character usually is sufficient. All editing commands also have synonyms defined which are entered without either the dot or the carriage return and are, in general, control characters. Some commands require arguments, such as the name of a file, the name of a tree, *etc.* These arguments can be given directly after the command or in response to ALOE prompts. When a prompt is given, a default value is shown enclosed in square brackets. A simple carriage return indicates that the default value should be used. Otherwise, the value entered is used as the argument.

The editing commands for an ALOE are described next. The synonyms for each command are shown in parentheses following the command name. The "$" symbol in synonyms stands for the *<escape>* character.

A.1. Cursor Movement

Cursor-in

._IN (<cursor-pad-down>)
Moves the cursor into the first legal offspring of the current node according to current unparsing scheme. **Cursor-in** automatically does a **cursor-next** if at a terminal or non-visible node.

Cursor-out

._OUT (<cursor-pad-up>)

Moves the cursor to the parent of the current node.

Cursor-next

._NEXT (<cursor-pad-right>)

Moves the cursor to the next sibling of the current node if one is defined according to the current unparsing scheme and the setting of the *cursor-follows* mode. If no sibling is defined, the cursor is then moved to the next sibling of the parent of the current node, recursively. If the current node is the last in the tree (as defined in pre-order) then the command has no effect.

Cursor-previous

._PREVIOUS (<cursor-pad-left>)

Moves the cursor to the previous sibling of the current node if one is defined according to the current unparsing scheme and the setting of the *cursor-follows* mode. If the current node is the leftmost node then the cursor is moved to the previous sibling of the parent of the current node. If the current node is the leftmost node in the tree (as defined in pre-order) then the command has no effect.

Cursor-home

._HOME (<cursor-pad-home>)

If the current node is not the root of the current window, **cursor-home** moves the cursor there Otherwise, it moves the cursor to the root of the previous context window.

Cursor-back

.BACK (↑b)

Moves the cursor back to its previous position, provided that the last command was a cursor moving command.

Find

.FIND <string> (↑f)

Searches the tree for a matching variable name, constant name, operator synonym, or operator name. The search is restricted to the current window. If no string is given one is prompted for. If a carriage return (<cr>) is typed for the string prompt the string specified in the previous search is used.

Rfind

.RFIND <string> (↑xb)

Searches the tree in reverse for a matching variable name, constant name, operator synonym, or operator name. The search is restricted to the current window. If no string is given one is prompted for. If a carriage return (<cr>) is typed for the string prompt the string specified in the previous search command is used.

Class

.CLASS <string> (↑x↑n)

Searches the tree for the first node in the specified class. The search is restricted to the current window. If no string is given one is prompted for. If a carriage return (<cr>) is typed for the string prompt the string specified in the previous search command is used.

Rclass

.RCLASS <string> (↑x↑p)

Searches the tree in reverse for a node in the specified class. The search is restricted to the current window. If no string is given one is prompted for. If a carriage return (<cr>) is typed for the string prompt the string specified in the previous search command is used.

First

.FIRST (↑x↑b)

If the current node is on a list, then the cursor is moved to the first item on the list. Otherwise the cursor is moved to the first sibling.

Last

.LAST (↑x↑l)

If the current node is on a list, then the cursor is moved to the last item on the list. Otherwise the cursor is moved to the last sibling.

Next

.NEXT (↑n)

Searches the tree for the next declaration or statement depending on your context. The target is set to a declaration as you enter a procedure and changed to a statement when you enter the statement list. The search is restricted to the current window.

Previous

.PREVIOUS (↑p)

Searches the tree for the previous declaration or statement depending on your context. The target is set to a declaration as you enter a procedure and changed to a statement when you enter the statement list. The search is restricted to the current window.

Numerical Arguments for Cursor Movement

.<number>

The explicit cursor moving commands (**cursor-in**, **cursor-out**, **cursor-next**, **cursor-previous**, and **cursor-home**) have an optional parameter that precedes them. The numerical argument indicates how many applications of the given command should be made. The argument is not a command in that it cannot be used alone.

A.2. Help Information

All help information available through these commands is displayed in the help window.

Operator Help

.HELP (↑x?)

If the current node is a meta node, .HELP displays the list of applicable language commands (and their synonyms). Otherwise, the list of editing commands is displayed.

Command Help

.? (.?)

Displays the list of editing commands (and their synonyms).

A.3. Tree Manipulation

Clip Subtree

.CLIP <tree-name> (↑k)

Clips current subtree into a named tree which is kept in the clipped area separate from the main tree. The name of the tree can be specified following the command or it will be prompted for.

Insert Subtree

.INSERT <tree-name> (↑x↑i)

Inserts a clipped subtree at the current node (which must be a meta node) provided that the root operator of the subtree is legal in this position. If no tree name is specified one is prompted for.

Extend List

.EXTEND (te)

Extends a list with a new meta node. If the current node is a list node (variable arity node) then an element is created at the beginning of the list. If the current node is a member of a list then the meta node is inserted immediately after it.

Extend List Backwards

.BEXTEND (txtb)

If the current node is a member of a list, it places a meta node immediately before the node. It is not applicable anywhere else.

Prepend to List

.PREPEND (txta)

If the current node is a member of a list, it places a meta node at the beginning of the list.

Append to List

.APPEND (txte)

If the current node is a member of a list, it places a meta node at the end of the list.

Delete

.DELETE (td)

Deletes the current subtree. If the subtree is an element of a fixed arity node, then a meta node is inserted in its place. If the subtree is an element of a list, the element is removed completely from the list.

Replace

.REPLACE (tr)

Deletes the current subtree. If the subtree is an element of a fixed arity node, then a meta node is inserted in its place. If the subtree is an element of a list, the element is replaced by a meta node of the appropriate class.

Nest

.NEST <operator name> (tn)

Takes the current subtree and nests it into a subtree that will have the operator as root operator. The operator name can be given following the command or it will be prompted for. A nesting that would result in an invalid tree is not permitted. If the new subtree has more that one

offspring, it finds the first match (independent of unparsing scheme) for the current subtree in the new subtree.

Transform

.TRANSFORM <operator name> (tt)

Transforms the operator of the current node to the desired one. For the transformation to succeed, the new operator must be in the same class as the old one and the respective offspring must also match exactly.

A.4. Input/Output

Read Program

.READPROG <file-name> (txtr)

Reads a tree from a file. Checks that the file contains a valid tree. Replaces the current tree with the new tree. Checks with the user if the current tree has not been saved. The file name can be given after the command or given to the ALOE prompt.

Load Tree

.LOADTREE <file-name> (txtv)

Loads a tree from a file into a clipped area. A clipped window is assigned to it with the name of the window taken from the file name. The name of the file can be given after the command or given to the ALOE prompt.

Write Tree

.WRITE (txtw)

Writes a tree into a file in tree form. The default prompt is the file name given at invocation of ALOE.

Unparse into File

.UNPARSE <file-name> (txtt)

Unparses the tree into a text file. The file name can be given after the command or to the ALOE prompt. Useful for producing printouts. Note that this command differs from .WRITE only in the form the written file takes.

A.5. Exit ALOE

Quit and Save

.QUIT (txtf)

Saves the current tree in a file in tree format and leaves ALOE. It uses the file name given at invocation.

Cancel

.CANCEL (tc)

Leaves ALOE. If the tree has been changed since the last **.WRITE** command, the user is warned and given a chance to abort the command.

A.6. Display Manipulation

Display Tree

.DISPLAY (tl)

The screen is cleared and redisplayed. Useful when operating system messages or other such noise gets displayed in the screen.

Window Down

.WDOWN (twtn)

Scrolls the tree window down by one half of the window length.

Window Left

.WLEFT (twta)

Scrolls the tree window left by one third of the window width.

Window Right

.WRIGHT (twte)

Scrolls the tree window right by one third of the window width.

Window Up

.WUP (twtp)

Scrolls the tree window up by one half of the window length.

Select Window

.WINDOW <window-name> (^w^w)

The selected window is displayed on the screen. The window name can be given after the command or to the ALOE prompt. The window must be a tree or a clipped window. If it is a tree window, the appropriate context switch takes place as if the cursor had been moved there explicitly.

A.7. Other Commands

Fork a UNIX Shell

.! (^x!)

Calls the UNIX shell from ALOE.

Edit

.EDIT (^x^t)

Invokes the text editor specified by the shell variable EDITOR, to edit a constant, long constant, or text node. Upon return the screen is updated to incorporate the edited string.

Set Mode

.MODE (^x^m)

Sets the mode to the new value. ALOE first prompts for the name of the mode and then for the new value of the mode.

Set Unparsing Scheme

.SCHEME <scheme-number> (^x^U)

This is a command to let the user change the current unparsing scheme. Takes as argument the number of the unparsing scheme. The scheme number can be specified after the command or given to the ALOE prompt. If the argument is out of range (larger that the largest defined unparsing scheme) then scheme zero is used.

Appendix B

Unparsing Scheme Commands

Each unparsing scheme is a string which has descriptions of the text to be used, the syntactic sugar to be used and the formatting style. The text given in the unparsing scheme is displayed as is. Only sequences starting with the "@" or "%" character are treated differentiy. The formatting commands availiable in unparsing schemes are:

@n	Insert a newline in the output.
@t or @>	Insert four spaces in the output.
@<	Go back four characters (stopping at the beginning of the line).
@+	Increase the indentation level (to take effect at the next "@n"). Every indentation is four spaces.
@-	Decrease the indentation level (to take effect at the next "@n").
@l	Flush left (start next piece of output at left margin of current line).
@h	Go back one character (provided it is not at the beginning of a line).
@b	Go back to previous line (undo "@n").
@u<n>	Change the current unparsing scheme to <n>. Push the current scheme index on a one-level stack.
@u	Reset unparsing scheme to value of the one-level stack.
@p<n>	Push marker <n> onto stack. Markers are used to "remember" column positions for formatting. They are specially useful when the desired formatting depends on the size of identifiers or strings. First marker is numbered zero.
@r<n>	Pop marker <n>.

`@g<n>` Get marker <n>. Moves the unparsing cursor to the column position specified by the marker.

`@@` Display an "@" character.

`@%` Display a "%" character.

The way the objects of the nodes are displayed is different for terminals and non-terminals. For terminals, the following unparsing commands are available:

`@c` Display value of character constant, string, text, integer, real, boolean, user node, or userblanks node.

`@s` Display variable name from symbol table.

The unparsing commands available for non-terminals are:

`@<n>` Unparse the <n>th offspring recursively. Used only for fixed arity nodes where the offspring are numbered from one on up. For example, "`while (@1)@+@n@2@-@n`" specifies that the node should be unparsed starting with the string "`while (`" followed by a recursive invocation of the unparser on the first offspring, a "`)`", a line break, the unparsing of the second offspring indented one level, and another line break. The order in which the offspring are unparsed can be different from the one specified in the abstract syntax. The "`n`" in "`<n>`" refers to the abstract syntax specification order. Finally, nodes can be hidden (made "non-visible") by simply omitting them in the unparsing scheme.

`<pr>@0<t>[@q<po>][@e<s>]`
Unparse the list node. Used only for non-terminals that are list nodes. The "`@0`" indicates that each element of the list should be unparsed in order. The "`<pr>`" is the prelude string that should be printed before the list is unparsed. The string "`<t>`" is used to separate list elements "`(<t>`" is terminated by either the following "`@q`" or the end of the unparsing scheme). The string "`<po>`" is the postfix that is printed after the list is unparsed. The optional "`@e`" indicates how the list should be unparsed if it is empty (*i.e.*, has no current elements). All of the strings may contain text and other unparsing commands. The parts in square brackets are optional. If no empty specification is given then nothing will be unparsed when the list is empty. For example, the scheme "`versions:@+@n@0;@n@q@-@nend@e<no versions>`" specifies that the list should be unparsed starting with the string "`versions:`", a line break and then the elements of the list separated by a "`;`" and a new line. The list should be terminated by a new line and the word "`end`" aligned with "[versions:]". If the list is empty then it should only unparse the string "`<no versions>`".

`@x` Used only for non-terminals that have filenodes associated with them and indicates that the subtree is not "visible".

`@z` Used in conjunction with "`@x`" to specify the place where the name of the filenode should be placed.

All the letters after the "`@`" in unparsing commands may be either upper or lower case. Additionally, with one exception, anywhere an "`@`" can occur a "`%`" can also be used. The exception is that in fixed arity operators, "`@<n>`" means that the node should be unparsed and visited (*i.e.*, you can stop the cursor there), while "`%<n>`" means to unparse but not to visit the node. In the case of lists it means that no element of the list can be visited.

Appendix C

ALOE Implementation Environment Routines

The ALOE implementation environment provides an environment for the implementor of action routines and extended commands. It provides a data encapsulation mechanism for the internal representation that defines the data structures that are accessible as well as the operations that can be performed on them. These operations provide the facilities for inspection, traversal and modification of the internal tree representation. They guarantee the syntactic correctness and integrity of the internal representation. The ALOE implementation environment actually provides an operational definition of the internal structure.

The remainder of this appendix is broken up into logical groups of routines. The specification for the routines is given using the GC [Feiler 79] format with the types of the parameters specified in the parameter list. A full and detailed explanation of these routines can be found in [Medina-Mora 81a].

C.1. Tree Manipulation Routines

This section describes routines that are used to manipulate nodes and subtrees of the tree.

```
int ismeta(struct tnode *node)
```
Check for meta.

```
struct tnode *findmeta(struct tnode *node)
```
Find meta.

```
struct tnode *chkmake(int opt;struct *thisnode·char *valuestr)
```

Validate and make a node.

```
struct tnode *chkcopy(struct tnode *source, *dest; int doact)
```
Copy a subtree.

```
struct tnode *delsubstree(struct tnode *thisnode)
```
Delete a subtree.

```
struct tnode *applyauto(struct tnode *thisnode;int doact)
```
Automatic application of operators.

```
  Walk(struct tnode *thisnode; int (*proc)())
```
Apply a function to each element of subtree.

```
RemAllMetas(struct tnode *list)
```
Remove all metas in list.

```
char terminal(int optype)
```
Get type of node.

```
int arity(int optype)
```
Get arity of node.

```
char *opname(int optype)
```
Get operator name.

```
struct tnode *getsysroot()
```
Get system root.

```
struct tnode *getroot()
```
Get root of window.

C.2. List Manipulation Routines

This section describes routines that are available for the manipulation of variable arity nodes (also referred to as list nodes) in ALOE. All these routines report an error "node is not a list" and return NIL whenever the pointer passed to it is not a list (or a member of a list when the parameter is supposed to be one).

```
struct tnode *addfirst(struct tnode *list)
```
Add meta to front of list.

```
struct tnode *addlast(struct tnode *list)
```
Add meta to end of list.

```
struct listnode *getlist(struct tnode *list)
```
Get list header.

```
int islist(struct tnode *list)
```
Check if list.

```
int lengthlist(struct tnode *list)
```
Get length of list.

```
int listindex(struct tnode *node)
```
Find element index.

```
listwalk(struct tnode *list; int (*proc) ())
```
Apply function tolist elements.

```
struct tnode *nthlist(struct tnode *list; int n)
```
Get nth. element.

```
struct listnode *searchlist(struct tnode *node)
```
Get header element.

```
struct listnode *cdr(struct listnode *cell)
```
Get next header.

C.3. Access Control Routines

This section describes the routines that are provided to control the commands and construction capabilities of an ALOE. The basic idea is that there is a stack of bit vectors that represent which commands are legal at various levels of the tree. The top of the stack indicates the current set of legal commands. The elements of this stack are of type `struct LegalAction *` (pointer to `LegalAction` structure, the details of the type structure are hidden). These elements should be created (by `topAction` or `newAction`) and pushed onto the stack whenever rights are changed (usually on an ENTRY action call) and popped off the stack when they are to be restored (usually on an EXIT action call).

```
initAction()
```
Initialize access stack.

```
struct LegalAction *newAction()
```
Create new access element.

```
restrict(char *cmd; struct LegalAction *curAction)
```
Restrict command.

```
permit(char *cmd; struct LegalAction *curAction)
```
Permit command.

```
consNOTOK(struct LegalAction *curAction)
```
Restrict construction.

```
consOK(struct LegalAction *curAction)
```
Permit construction.

```
pushAction(struct LegalAction *curAction)
```
Push access element onto stack.

```
popAction()
```
Pop access element.

```
struct LegalAction *topAction()
```
Take top access element.

C.4. Error Reporting Routines

This section describes routines that help the writer of action routines with Interfacing to the error reporting mechanisms provided by ALOE. Errors, warnings, and plain messages are queued in a buffer. Upon return from an action routine or an extended command, ALOE will display all the messages in order.

```
syserror(char *msg)
```
Indicate a system error.

```
sysabort()
```
Abort the system.

```
error(char *msg; struct tnode *enode)
```
Queue an error.

```
warn(char *msg; struct tnode *wnode)
```
Queue a warning.

```
message(char *msg; struct tnode *mnode)
```
Queue a message.

```
int errcnt()
```
Get error count.

C.5. Filenode Routines

This section describes the routines needed to manipulate filenodes (a filenode is often referred to as a `tnodef`). Filenodes are used to "partition" the internal tree into a database of separate files. The routines listed here provide all the necessary manipulation of file databases. In this section, *context* is the smallest subtree that the current node is in, where the root of the subtree is a FNONTERMINAL (a non-terminal node with a filenode associated with it. A context stack is automatically kept by an ALOE. Elements are pushed on to the stack as contexts are entered and popped off when they are exited.

```
int istnodef(struct tnode *node)
```
Check for filenode.

```
struct tnodef *curcontext()
```
Get current context.

```
cntxdirty()
```
Dirty current context.

```
struct tnode *getfather(struct tnode *thisnode)
```
Get parent.

```
struct tnode *tofather(struct tnode *thisnode)
```
Get father, save context.

```
struct tnode *getson(struct tnode *thisnode; int wson)
```
Get offspring.

```
struct tnode *toson(struct tnode *thisnode; int wson)
```
Get offspring, save context.

```
struct tnode *getcar(struct listnode *header)
```
Get list element.

```
struct tnode *tocar(struct listnode *header)
```
Get list element, save context.

```
struct tnodef *gettnodef(struct tnode *thisnode)
```
Get parent of FNONTERMINAL.

```
struct tnode *getfson(struct tnodef *thisnode)
```
Get Offspring of filenode.

```
struct tnodef *findfnode(struct tnode *node)
```
Find filenode above node.

```
CheckPoint(struct tnode *node)
```
Checkpoint a subtree.

C.6. Status Manipulation Routines

This section describes the routines provided for manipulation of the status fields of tree nodes. They may be used by an ALOE implementor in cases where the status checking needed for trees is relatively simple. In other cases, the implementor should write and use more complex status schemes.

All the routines in this section access the status field by using the `tnstat` type definition. The two fields manipulated are `actstat` and `count`. The `actstat` field can contain three values: OK, NOTOK, and UNK. This indicates the current status of the `tnode` (UNK indicates

the status is currently unknown). The count field is the number of offspring of the node that have a status of OK. The routines described below manipulate these fields with these semantics in mind.

```
initstat(struct tnode *node)
```
Initialize status.

```
statusOK(struct tnode *node)
```
Set status OK.

```
statusNOTOK(struct tnode *node)
```
Set status not OK.

```
int checkit(struct tnode *node; int nsons)
```
Check for status.

C.7. Window Manipulation Routines

This section describes the routines needed to manipulate the windows on the screen.

```
struct window *LkupWindow(char *windname)
```
Look up window.

```
AsgTextWindow(struct window *wind)
```
Assign text window.

```
AsgTreeWindow(struct window *wind; int scheme; struct tnode *rnode;
              struct tnode *cnode)
```
Assign tree window.

```
NewTWindow(struct tnode *rootnode; int scheme; char *wname)
```
New tree window.

```
struct tnode *RemTWindow()
```
Release tree window.

```
struct window *SetWindow(struct window *wind)
```
Set current window.

```
RelWindow(struct window *wind)
```
Release window.

```
int setoutwind(char *wname; integer clear)
```
Set output window.

```
resetoutwind()
```
Reset output window.

C.8. Miscellaneous Routines

```
changedtree()
```
Indicate that the tree has changed.

```
struct tnode *callonpath(struct tnode *fromnode, *tnode)
```
Call action routines on path.

References

[Alberga 81] Alberga, C. N., Brown, A. L., Leeman, G.B. Jr., Mikelsons, M. and Wegman, M. N.
A Program Development Tool.
In *Proceedings of the Eight Annual ACM Symposium on Principles of Programming Languages*, pages 92-104. January, 1981.

[Albrecht 80] Albrecht, P.F.
Source-to-Source Translation: Ada To Pascal and Pascal to Ada.
Sigplan Notices 15(11), Nov, 1980.

[Archer 81a] Archer, James E. Jr, Conway, Richard and Schneider, Fred B.
User Recovery and Reversal in Interactive Systems.
Technical Report TR 81-476, Cornell University, Department of Computer Science, October, 1981.

[Archer 81b] Archer, James E. Jr. and Conway Richard.
COPE: A Cooperative Programming Environment.
Technical Report TR 81-459, Cornell University, Department of Computer Science, June, 1981.

[Backus 59] Backus, J. W.
The Syntax and Semantics of the Proposed International Algebraic Language of the Zurich ACM-GAMM Conference.
In *Proceedings of the International Conference on Information Processing*, pages 125-132. UNESCO, 1959.

[Ball 80] Ball, J. Eugene.
Alto as Terminal.
1980.
Internal Documentation. Carnegie-Mellon University, Department of Computer Science.

[Ball 81] Ball, J. Eugene.
Canvas: Graphics for the Spice Personal Timesharing System.
In *Proceedings of Comgraph 1981*. Online Conferences, October, 1981.

[Barstow 81] Barstow, David R.
Overview of a Display-Oriented Editor for Interlisp.
In *Proceedings of the Seventh International Joint Conference on Artificial Intelligence*, pages 927-929. August, 1981.

[Conway 76] Conway, R. and Constable, R.
PL/CS -- A Disciplined Subset of PL/I.
Technical Report TR 76-293, Cornell University, Department of Computer Science, 1976.

[Demers 81] Demers, A., Reps, T. and Teitelbaum, T.
Incremental evaluation for attribute grammars with application to syntax-directed editors.
In *Conference Record of the Eighth Annual ACM Symposium on Principles of Programming Languages*, pages 105-116. ACM, January, 1981.

[DeRemer 76] DeRemer, Frank and Kron, Hans H.
Programming-in-the-Large vs Programming-In-the-Small.
IEEE Transactions on Software Engineering , June, 1976.

[Deutsch 80] Deutsch, L.P. and Taft, E.A. (editors).
Requirements for an Experimental Programming Environment.
Technical Report CSL 80-10, Xerox Palo Alto Research Center, June, 1980.

[DoD 80] United States Department of Defense.
Reference Manual for the Ada Programming Language.
1980.
Proposed Standard Document.

[Dolotta 76] Dolotta, T. A. and Mashey, J. R.
An Introduction to the Programmer's Workbench.
In *Proceedings of the Second International Conference on Software Engineering*, pages 164-168. ACM & IEEE, 1976.

[Donzeau-Gouge 80]
Donzeau-Gouge, Veronique, Huet, Gerard, Kahn, Gilles and Lang, Bernard.
Programming Environments Based on Structured Editors: The MENTOR Experience.
Presented at the Workshop on Programming Environments in Ridgefield, CT on June 1980.

[Feiler 79] Feiler, Peter H. and Medina-Mora, Raul.
The GC Language.
1979.
Gandalf Internal Documentation. Carnegie-Mellon University, Department of Computer Science.

[Feiler 81] Feiler, Peter H.
The Gandalf Display Package.
1981.
Gandalf Internal Documentation. Carnegie-Mellon University, Department of Computer Science.

[Feiler 82a] Feiler, Peter H.
A Language-Oriented Interactive Programming Environment Based on Compilation Technology.
PhD thesis, Carnegie-Mellon University, Department of Computer Science, 1982.

[Feiler 82b] Feiler, Peter H.
A GC to PASCAL translator using ALOE.
1982.
Private Communication.

[Feldman 79] Feldman, Stuart I.
Make - A Program for Maintaining Computer Programs.
Software - Practice and Experience 9(3):255-265, March, 1979.

[Ganzinger 77] Ganzinger, H., Ripken, K. and Wilhelm, R.
Automatic generation of optimizing multipass compilers.
In B. Gilchrist (editor), *Information Processing 77, Proceedings of IFIP Congress 77*, pages 535-540. North-Holland, 1977.

[Gosling 81a] Gosling, James.
Unix Emacs
Carnegie-Mellon University, Department of Computer Science, 1981.

[Gosling 81b] Gosling, James.
A Redisplay Algorithm.
In *Proceedings of the ACM SIGPLAN/SIGOA Symposium on Text Manipulation*, pages 123-129. ACM SIGPLAN/SIGOA, June, 1981.

[Graham 79] Graham, Susan L., Haley, Charles B. and Joy, William N.
Practical LR Error Recovery.
In *Proceedings of the ACM SIGPLAN Symposium on Compiler Construction*, pages 168-175. ACM SIGPLAN, August, 1979.

[Habermann 79a] Habermann, A. Nico.
A Software Development Control System.
1979.
Internal Documentation. Carnegie-Mellon University, Department of Computer Science.

[Habermann 79b] Habermann, A. Nico.
The Gandalf Research Project.
In *Computer Science Research Review 1978-79*, pages 28-35. Carnegie-Mellon University, Department of Computer Science., 1979.

[Habermann 80] Habermann, A. Nico.
Notes on Programatics and its Language Alfa.
1980.
Private Communication.

[Habermann 82] Habermann, A. Nico and Notkin, David S.
The Gandalf Software Development Environment.
1982.
Carnegie-Mellon University. Submitted for Publication.

[Hansen 71] Hansen, Wilfred J.
Creation of Hierarchic Text with a Computer Display.
PhD thesis, Stanford University, Department of Computer Science, June, 1971.

[HDS 79] Human Designed Systems, Inc.
Concept Reference Manual.
Human Designed Systems, Inc, 1979.

[Ingalls 78] Ingalls, Daniel H. H.
The Smalltalk-76 Programming System Design and Implementation.
In *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages*. ACM, January, 1978.

[Ivie 77] Ivie, Evan L.
The Programmer's Workbench - A Machine for Software Development.
CACM 20(10), Oct, 1977.

[Jensen 74] Jensen, K. and Wirth, N.
Pascal User Manual and Report.
Springer-Verlag, 1974.

[Johnson 75] Johnson, S. C.
YACC - Yet Another Compiler-Compiler.
Technical Report Computer Science 32, Bell Laboratories, July, 1975.

[Johnson 82] Johnson, Gregory F. and Fischer, Charles N.
Non-syntactic Attribute Flow in Language Based Editors.
In *Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages*, pages 185-195. ACM, January, 1982.

[Kaiser 81] Kaiser, Gail E.
The Gandalf Symbol Table.
1981.
Gandalf Internal Documentation. Carnegie-Mellon University, Department of Computer Science

[Kaiser 82] Kaiser, Gail E. and Habermann, A. Nico.
An Environment for System Version Control.
1982.
Carnegie-Mellon University, Department of Computer Science. Submitted for Publication.

[Kernighan 78] Kernighan, B. W. and Ritchie, D. M.
Prentice-Hall Software Series: The C Programming Language.
Prentice-Hall, 1978.

[Knuth 68] Knuth, Donald E.
Semantics of Context-Free Languages.
Mathematical Systems Theory 2(2):127-145, 1968.

[Lamb 82] Lamb, David A.
IDL Translator Design Document.
1982.
Internal Documentation. Carnegie-Mellon University, Department of Computer Science.

[Lampson 79] Lampson, Butler.
Bravo Manual
XEROX Palo Alto Research Center, 1979.

[Medina-Mora 81a]
Medina-Mora, Raul and Notkin, David S.
ALOE Users' and Implementors' Guide.
Technical Report CMU-CS-81-145, Carnegie-Mellon University, Department of Computer Science, November, 1981.

[Medina-Mora 81b]
Medina-Mora, Raul and Feiler, Peter H.
An Incremental Programming Environment.
IEEE Transactions on Software Engineering SE-7(5):472-482, September, 1981.
Revised version of the paper presented at the Fifth International Conference on Software Engineering, San Diego, march 1981.

[Mikelsons 80] Mikelsons, M. and Wegman, M. N.
PDE1L: The PL1L Program Development Environment Principles of Operation.
Technical Report RC 8513, Computer Science Department, IBM T. J. Watson Research Center, Yorktown Heights, NY, September, 1980.

[Mikelsons 81] Mikelsons, Martin.
Prettyprinting in an Interactive Programming Environment.
In *Proceedings of the ACM SIGPLAN/SIGOA Symposium on Text Manipulation*, pages 108-116. ACM SIGPLAN/SIGOA, June, 1981.

[Mitchell 78] Mitchell James G., Maybury, William and Sweet, Richard.
Mesa Language Manual
XEROX Palo Alto Research Center, 1978.

[Morris 81] Morris, Joseph M. and Schwartz, Mayer D.
The design of a language-directed editor for block-structured languages.
In *Proceedings of the ACM SIGPLAN/SIGOA Symposium on Text Manipulation*, pages 28-33. ACM SIGPLAN/SIGOA, June, 1981.

[Nestor 81] Nestor, John R. and Beard, Margaret A.
Front End Generator System, User's Guide.
July, 1981.
Internal Documentation. Carnegie-Mellon University, Department of Computer Science.

[Notkin 81a] Notkin, David S.
On the specification of Access Control.
1981.
Private Communication.

[Notkin 81b] Notkin, David S.
On the use of the term: programming-in-the-many.
1981.
Private Communication.

[Notkin 82a] Notkin, David S.
Implementation of a small mail system using ALOE.
1982.
Private Communication.

[Notkin 82b] Notkin, David S. and Kaiser, Gail E.
The Implementation of the Gandalf Software Development Environment.
1982.
Carnegie-Mellon University, Department of Computer Science. To appear.

[Notkin 82c] Notkin, David S.
On the implementation of *Aloegen*.
1982.
Private Communication.

[Notkin 82d] Notkin, David S.
On the Implementation of an ALOE for SCRIBE.
1982.
Private Communication.

[Oppen 79] Oppen, Derek C.
Pretty Printing.
Technical Report STAN-CS-79-770, Stanford University, Computer Science Department, October, 1979.

[Perdue 74] Perdue, Crispin.
User's Introduction to UCI Lisp.
Technical Report, Carnegie-Mellon University, Department of Computer Science, August, 1974.

[Rahia 77] Rahia, K. J.
On attribute grammars and their use in a compiler writing system.
Technical Report A-1977-4, Department of Computer Science, University of Helsinki, August, 1977.

[Reid 80] Reid, Brian K. and Walker, Janet H.
SCRIBE Introductory User's Manual
Unilogic, Ltd, 1980.

[Reps 81] Reps, Thomas.
The Synthesizer Editor Generator: Reference Manual
Internal Documentation. Cornell University, Department of Computer Science, 1981.

[Reps 82] Reps, Thomas.
Optimal-time incremental semantic analysis for syntax-directed editors.
In *Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages*. ACM, January, 1982.

[Ritchie 74] Ritchie, Dennis M.
The Unix Time-Sharing System.
Communications of the ACM 17(7):365-375, July, 1974.

[Sproull 79] Sproull, Robert F.
Raster Graphics for Interactive Programming Environments.
Technical Report CSL-79-6, XEROX Palo Alto Research Center, June, 1979.

[Stallman 81] Stallman, Richard M.
EMACS, the extensible, customizable, self documenting display editor.
In *Proceedings of the ACM SIGPLAN/SIGOA Symposium on Text Manipulation*, pages 147-156. ACM SIGPLAN/SIGOA, June, 1981.

[Swinehart 74] Swinehart, D. C.
Copilot: A Multiple Process Approach to Interactive Programming Systems.
PhD thesis, Stanford University, July, 1974.

[Teitelbaum 81a] Teitelbaum, Tim and Reps, Thomas.
The Cornell Program Synthesizer: A Syntax-Directed Programming Environment.
Communications of the ACM 24(9):563-573, September, 1981.

[Teitelbaum 81b] Teitelbaum, Tim, Reps, Thomas and Horowitz, Susan.
The why and wherefore of the Cornell Program Synthesizer.
In *Proceedings of the ACM SIGPLAN/SIGOA Symposium on Text Manipulation*, pages 8-16. ACM SIGPLAN/SIGOA, June, 1981.

[Teitelbaum 82] Teitelbaum, Tim.
Transformations in the Synthesizer.
1982.
Private Communication.

[Teitelman 77] Teitelman, Warren.
A Display Oriented Programmer's Assistant.
In *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, pages 905-915. August, 1977.

[Teitelman 78] Teitelman, Warren.
Interlisp Reference Manual
XEROX Palo Alto Research Center, 1978.

[Tichy 82] Tichy, Walter F.
A Data Model for Programming Support Environments and its Applications.
In Schneider, H.-J. and Wasserman A. (editor), *Automated Tools for Information Systems Design*, pages 31-48. IFIP, North-Holland, January, 1982.